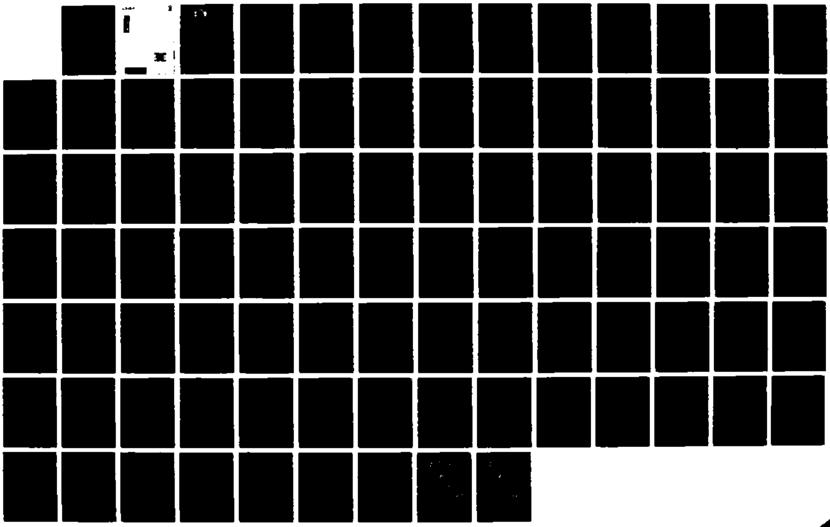
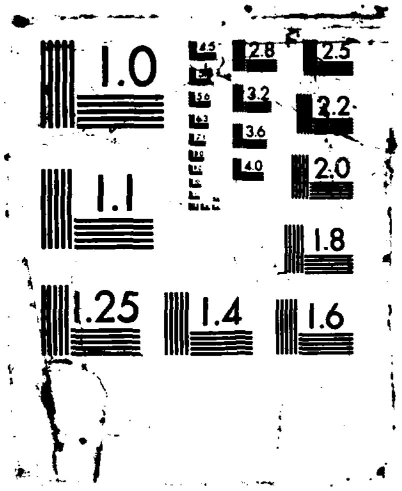
AD-A192 723 CONSTRUCTING A HIGHLY-AVAILABLE LOCATION SERVICE FOR A DISTRIBUTED ENVIRO..(U) MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE.. D J HWANG JAN 88 1/1

UNCLASSIFIED MIT/LCS/TR-410 N00014-83-K-0125 F/G 12/7 NL




1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

3

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

DTIC ELECTE MAR 0 3 1988 S H D

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-410	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-83-K-0125
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science	6b. OFFICE SYMBOL *(If applicable)*
7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Dept. of Navy	
6c. ADDRESS *(City, State, and ZIP Code)* 545 Technology Square Cambridge, MA 02139	7b. ADDRESS *(City, State, and ZIP Code)* Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD	8b. OFFICE SYMBOL *(If applicable)*
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS *(City, State, and ZIP Code)* 1400 Wilson Blvd. Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
CONSTRUCTING A HIGHLY-AVAILABLE LOCATION SERVICE FOR A DISTRIBUTED ENVIRONMENT

12. PERSONAL AUTHOR(S)
Hwang, Deborah Jing-Hwa

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15 PAGE COUNT
Technical	FROM ____ TO ____	1988 January	86

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
distributed systems, highly-available, location service, multipart timestamps, reconfiguration, replacement, replication

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

One possible advantage a distributed system has over a centralized system is the ability to move objects from one node to another. For example, we may want to move an object if the node where it resides is overloaded. This thesis proposes to use a location service to aid in finding objects that move. The service is highly-available; it will tolerate system failures like node crashes and network partitions without shutting down completely. The service is also efficient; the response time of the service is reasonable, and it does not increase the number and sizes of messages excessively.

We achieve high availability and efficiency by replicating the service state. The replication technique we have chosen is a new method, the multipart timestamp technique that is based on multipart timestamps and gossip messages. This technique provides us with higher availability and efficiency than traditional replication techniques without sacrificing consistency. We also extend this technique to allow reconfiguration.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b TELEPHONE *(Include Area Code)* (617) 253-5894 — 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

Unclassified

Constructing a Highly-Available Location Service for a Distributed Environment

by

Deborah Jing-Hwa Hwang

November 1987

This research was supported by an Office of Naval Research Graduate Fellowship, and in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office Naval Research under contract N00014-83-K-0125 and the National Science Foundation under grant DCR-8503662.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

Constructing a Highly-Available Location Service for a Distributed Environment

by

Deborah Jing-Hwa Hwang

Submitted to the
Department of Electrical Engineering and Computer Science
on November 18, 1987, in partial fulfillment of the requirements
for the Degree of Master of Science

Abstract

One possible advantage a distributed system has over a centralized system is the ability to move objects from one node to another. For example, we may want to move an object if the node where it resides is overloaded. This thesis proposes to use a *location service* to aid in finding objects that move. The service is *highly-available*; it will tolerate system failures like node crashes and network partitions without shutting down completely. The service is also *efficient*; the response time of the service is reasonable, and it does not increase the number and sizes of messages excessively.

We achieve high availability and efficiency by replicating the service state. The replication technique we have chosen is a new method, the *multipart timestamp technique* that is based on *multipart* timestamps and *gossip* messages. This technique provides us with higher availability and efficiency than traditional replication techniques without sacrificing consistency. We also extend this technique to allow reconfiguration.

Thesis Supervisor: Barbara H. Liskov
Title: NEC Professor of Software Science and Engineering

Keywords: distributed systems, highly-available, location service, multipart timestamps, reconfiguration, replacement, replication

Acknowledgments

First, I thank my advisor, Barbara Liskov, for suggesting this interesting topic and providing me with guidance in formulating and organizing the ideas in this thesis. I also thank the Office of Naval Research for their generous support.

I thank the members of the Programming Methodology, Theory of Distributed Systems, and Systematic Program Development groups for their support and encouragement. In particular, I thank my officemate, Brian Oki, for many enlightening discussions, technical or otherwise, and Gary Leavens for many helpful comments on this thesis. Paul Johnson and Dorothy Curtis get special thanks for keeping the Argus system running and answering my many questions about the system.

I thank my parents and my sisters, Bess and Amy. I could not ask for a family that provided more support and encouragement for my endeavors. They have all my love and respect.

I thank the Ashdown House community for their warmth and friendship. I especially want to thank Wil Howitt (the "Incorrigible Punster") for being a wonderful friend; Kathryn Hess for being a great friend and co-conspirator; Mike Ciholas for sharing his unique outlook on life; and those crazy Canadians, Michael Corvin and Donald Russell, for making me laugh when I least expected it and most needed it. They have made being here much more pleasant as well as adventurous.

Finally, I thank my old and dear friends from Iowa State who kept me crazy all those years and still try their best even now. Special thanks go to Susan Clarke for being the best friend I've ever had; Fred Weiser for just being himself; and Della Leonard for giving me a place to go when I wanted to get away (especially at Thanksgiving).

Table of Contents

Chapter One: Introduction 7

Chapter Two: Background 10

2.1 Model of computation 10
2.2 Argus 10
2.2.1 Guardians and handlers 11
2.2.2 Atomic actions and atomic objects 12
2.2.3 Mutex objects 12
2.2.4 System implementation 13
2.3 Moving objects in Argus 13

Chapter Three: A Highly-Available Location Service 16

3.1 Multipart timestamp technique 17
3.2 Operations of the location service 19
3.2.1 Specifications 19
3.2.2 An example 23
3.3 Abstract implementation 26
3.3.1 Data structures 26
3.3.2 Operations 28
3.3.3 Gossip 30
3.3.4 Garbage collection 32
3.3.5 Replica recovery 33
3.4 Clients of the location service 35
3.4.1 Guardian manager 35
3.4.2 Replacement system 36
3.4.3 Regular guardians 37
3.4.4 Discussion 38
3.5 Summary 39

Chapter Four: Implementing the Location Service in Argus 40

4.1 Serial solution 42
4.1.1 Data structures 43
4.1.2 Replica processing 44
4.2 Alternate solutions 46
4.2.1 Pre-commit solutions 47
4.2.1.1 Local multipart time solution 48
4.2.1.2 Prepare-phase solution 49
4.2.2 Post-commit solutions 49
4.2.2.1 Commit-phase solution 50
4.2.2.2 Non-transaction solution 50
4.3 Conclusions 51
4.3.1 Evaluation of serial solution 51

4.3.2 Comparison to alternative solutions 51

Chapter Five: Reconfiguration of the Server State 53

5.1 Extensions to support reconfiguration 54
5.1.1 Identifying replicas 56
5.1.2 Version numbers 56
5.1.3 Gossip processing in the hybrid state 57
5.1.4 Returning to normal processing and destroying obsolete replicas 61
5.2 Abstract implementation 61
5.2.1 Data structures 62
5.2.2 Changing configurations 62
5.2.3 Gossip processing 65
5.2.4 Other replica processing 67
5.2.5 Returning to normal processing and destroying obsolete replicas 69
5.3 An optimization 70
5.4 How clients find reconfigured services 71

Chapter Six: Conclusions 74

6.1 Evaluation of the multipart timestamp technique 75
6.2 Related work 78
6.2.1 Forwarding addresses 79
6.2.2 Searching 80
6.2.3 Establishing a rendezvous 81
6.3 Future work 82

Table of Figures

Figure 3-1:	A service state represented as a graph	22
Figure 3-2:	An example of using the location service	24
Figure 3-3:	State of a service replica	27
Figure 4-1:	Transaction scenario	41
Figure 4-2:	State of an implemented service replica	44
Figure 5-1:	State of a reconfigurable replica	63

Chapter One

Introduction

As the cost of computers decreases, finding ways to use multiple computers has been on the increase. One way to use multiple computers is to distribute parts of a computation over a system of nodes connected by a communications network. We call such a system a *distributed system*. A distributed system can offer several advantages over a traditional, centralized system. One possible advantage is the ability to move the objects of a system from one node to another. We might want to move an object for various reasons. For example, we may be able to increase system efficiency. If the cost of sending a message to an object is proportional to the length of the path from the sender to the object, then if the object is moved "closer" to the processes that manipulate it frequently, we would expect greater efficiency. Another example is to redistribute the message traffic or load at a node by moving some of the objects from the heavily used node to a node with less traffic or load. A third example is to prevent major disruptions in access. If a node is going to be down for a significant amount of time, we may want to move the objects residing on that node to another node. Or it may be that the node simply cannot maintain the object any longer; this might happen if the node is going to be removed from the system, or reassigned to another task.

Historically, objects were created, used, and destroyed at a single node. In such a scheme, the location of an object can be part of its name, so finding an object is simple; each node just "knows" how to determine where an object resides from its name. However, in a world where objects move, the problem of locating them becomes non-trivial. We can no longer just embed an object's location in its name. Since an object may no longer be at the node where it was last accessed, we must have a way of finding out its new location. Various methods for locating objects have been proposed. Forwarding addresses [4] can be used to allow processes to find objects that have moved by following a chain of addresses to the current location. A search for an object can be done using a broadcast technique [7].

This thesis investigates the use of a *location service* to aid in finding objects that move. Abstractly, a location service maintains associations of object names to locations. It provides operations to read and update these associations. These operations are the only way to interact with the service. Entities that invoke the service's operations are called *clients* of the service.

We have two main goals for our location service. First, it should be *highly-available*. Our definition of availability of query operations (operations that request information from the service) is if an object resides at a node that is accessible to the entity trying to use it, then with high probability the entity should be able to locate that object. We define the availability of update operations (operations that change the service state) as the probability that an operation invocation will be completed in a short period of time. To achieve this goal, the service must tolerate system failures like node crashes and network partitions. By tolerate, we mean that the service should degrade gracefully in the presence of continuing failures. It may not be able to provide full service, but it should provide as much information as possible and not just shut down completely.

The location service should also be *efficient*. Unfortunately, efficiency and availability are somewhat inversely related. Making the service highly-available may introduce inefficiencies. However, there are many techniques for achieving high availability. We want to choose one that does not severely affect the response time of the service, nor increase the number and sizes of messages excessively.

Simple, centralized implementations of a location service are easily realized and are very efficient, but do not meet our availability requirements. If the node where the service resides goes down, the service would be inaccessible to the entire system. If this node is down long enough, it could cause the system to stop functioning. Replication is a standard technique for achieving both high availability and efficiency. Multiple copies of the service state are kept at different nodes. Replication increases availability; if one replica is temporarily inaccessible, work can continue using a different replica. Replication can enhance performance by permitting clients to use the most easily accessible replica, but it can also decrease performance if several replicas are needed to perform an operation.

Once the data is replicated at several nodes, the information at different replicas may not be identical. This may occur if a node is down or inaccessible when an update is performed. When such inconsistencies happen, there must be some way to determine the correct response. One method is Gifford's majority voting algorithm [5]. In his scheme, the set of replicas visited by operations that read the service state must intersect with those that are visited by operations that modify the state. There is some flexibility in choosing the sets of replicas for each operation. For example, if the service state had three replicas, information could be written to three replicas and read from just one, or information could be written and read at two replicas. The first choice gives more availability and faster response time to read operations, while the availability of the write operation is poor, since all three replicas have to be accessible. The second choice trades off the availability and efficiency of read operations with the availability of write operations. We would like to make both types of operations available and efficient.

The replication technique we have chosen for our server is Liskov's multipart timestamp technique [10]. This is a new replication method based on *multipart* timestamps and *gossip* messages. We chose the multipart timestamp technique because it affords us higher availability and greater efficiency than more traditional replication techniques. It allows us to do reads and updates at any one replica.

The contributions of this thesis are:

- a location service for Argus and a basis for object finding in general
- a practical application of the multipart timestamp technique
- extension of the multipart timestamp technique to allow reconfiguration of the service state

In the remainder of this thesis, we present the design and implementation of a highly-available, efficient location service for the Argus system. Chapter 2 gives the context for our work. It states our model of computation: the assumptions we make about system hardware and failures. It describes the features of the Argus system that are relevant to this thesis. Also, it discusses what it means to move objects in Argus and where the location service fits into the system.

Chapter 3 lays out the basic design for the highly-available location service, ignoring implementation issues like transactions to simplify the presentation. We assume that all replacement transactions commit. This assumption is relaxed in Chapter 4, which discusses the interaction of the server with the transaction system. The changes to the replica state and operations to handle the issues raised by aborting transactions are presented as we describe an actual implementation of the service. Some alternate solutions are also discussed and compared with the implemented solution.

As the configuration of a system changes, we may need to change the configuration of the location service as well. We would like to be able to change the number or locations of the replicas that make up the server. Chapter 5 discusses extensions to the multipart timestamp technique to allow reconfiguration. We also address the issue of clients finding the service after a reconfiguration.

Chapter 6 presents our conclusions. We evaluate the multipart timestamp technique, compare the work in this thesis to some related work, and suggest some areas for future work.

Chapter Two

Background

In this chapter, we present background information to give a context for our work. First, we describe our model of computation: what our assumptions are and what our notion of failure is. Next, we describe the parts of the Argus system relevant to this thesis. Finally, we describe what it means to move objects in Argus and how the location service fits into the system.

2.1 Model of computation

Our model of computation makes the following assumptions about the system hardware and the effects of failures. A distributed system is a collection of physical nodes interconnected (only) by a communication network. These nodes may be in geographically distinct locations and may be administered by independent entities. A node may consist of one or more processors and any number of peripheral devices. We assume that processors are fail-stop [25]. Nodes can crash, but we assume that processors do not exhibit Byzantine behavior.

The communication network may be of arbitrary topology, perhaps a combination of local area networks and long haul networks. Processes on different nodes can communicate only by passing messages over the network. We assume that in the absence of network failures, any node can communicate with any other node. Again, we assume that there are no Byzantine failures, but otherwise the network can behave arbitrarily badly. In particular, it can partition. It can lose, delay, duplicate, or garble messages. It can deliver messages out of order. We assume that we can detect garbled messages and throw them away, treating them as lost messages. We do not assume that we can distinguish between delays caused by a node failure, a network failure, or a heavily loaded system.

2.2 Argus

Argus is both a programming language and a run-time system. The Argus programming language is a high-level approach for writing distributed applications that need reliable, consistent storage, but do not have severe real-time constraints [19]. The language is based on CLU [15] and is fully described in [16]. In particular, Argus retains all of the abstraction mechanisms of CLU [14] and adds several more. The main features of Argus that we are most interested in are the following: guardians and handlers, atomic actions and atomic objects, mutex objects, and some parts of the system implementation, namely handler names and guardian managers.

2.2.1 Guardians and handlers

The programming language Argus supports a module type called a *guardian.* A guardian is an abstraction of a processor and its state and resources. It may be thought of as a logical node of the system. A guardian encapsulates long-lived, resilient data and processes that manipulate this data. A guardian is created dynamically and resides wholly on one physical node. There may be more than one guardian at a physical node. Unlike regular objects, a guardian is an active entity; it has multiple internal processes. A guardian has zero or more background processes and a set of operations called *handlers.* Handlers may be thought of as procedure objects that reside at a guardian through which one can manipulate the guardian state and resources. Handlers are invoked as remote procedure calls, and their arguments and results are transmitted by value [9]. The only way to use a guardian is to call one of its handlers. Each handler call is run as a separate process within the guardian. A handler call has "at-most-once" semantics. If the call returns, it has been done at most once. If the call does not return or fails, it has (effectively) not been done at all.

Guardians are the unit of failure in Argus. They either function correctly (as programmed), or they fail completely. When a guardian fails, it is said to have *crashed.* The Argus system periodically tries to restart crashed guardians. When a guardian crashes, all of its processes die. A guardian may crash even if its node does not, but when a physical node crashes, all of its guardians crash as well.

A guardian is implemented by a guardian definition. This definition includes one or more operations called *creators* (used to create new guardian instances), the names of the guardian's handlers, the declaration of the guardian's state, the background code, and the recover code. A guardian has two types of state: *volatile* state is cheap, fast, and does not survive crashes; *stable* state is expensive and slower. It is stored on stable storage devices so that information will not be lost with arbitrarily high probability [12] when a guardian crashes. A guardian's state is initialized as part of its creation. When a guardian is restarted after a crash, its stable state is restored automatically and the recover code (if any) is run to reinitialize its volatile state. The background code is executed as a separate process after the creation of the guardian is completed and after every recovery from a crash. A guardian can be destroyed in only two ways. The guardian itself can execute a terminate statement, or it can be destroyed using a system program.

2.2.2 Atomic actions and atomic objects

Distributed computations in Argus are organized as *atomic actions* (or *transactions*). Transactions have two properties of interest. The first property is *indivisibility*. The execution of one transaction never appears to overlap or contain the execution of another transaction even if they run concurrently. The second property is *recoverability*. Either all of the effects of a computation are visible (the transaction has *committed*) or none of the effects are visible (the transaction has *aborted*). When a guardian crashes, all uncommitted transactions are aborted. Argus also supports nested subactions as a method of concurrency control and isolation of failures [21]. In particular, an action invoking a handler creates a subaction to send the call (the *call* action) and a subaction at the remote site to do the call. This allows the program to abort a handler invocation by aborting the call action without aborting the invoking action or waiting for the remote action to terminate.

Argus programs achieve atomicity through the sharing of *atomic objects*. Argus provides special built-in atomic objects as part of the language definition. These objects are synchronized by the system using strict two-phase read/write locking [2]. Recoverability is provided by keeping versions and a two-phase commit protocol [6]. A detailed description of the Argus recovery algorithm is given by Oki [23].

2.2.3 Mutex objects

Argus also supports a built-in type generator called *mutex*. Mutex objects are not necessarily atomic objects, although they can be shared by transactions. A mutex object is a mutable container with a lock associated with it. Mutually exclusive access to the contained object is obtained using the seize statement that acquires the mutex's lock in an exclusive mode.

Mutex objects are used primarily to implement user-defined atomic types. User-defined atomic types are abstract atomic types implemented by the user, usually to gain concurrency not allowed by the built-in atomic types. Like built-in atomic types, user-defined atomic types must provide indivisibility and recoverability mechanisms. Generally, a user-defined atomic type is implemented as a mutex surrounding a non-atomic collection of atomic objects. (See Weihl and Liskov's paper [28] for the theory of atomic objects and example implementations.)

2.2.4 System Implementation

When guardians are created, they receive unique identifiers (*guardian ids*) that serve as their system names. A guardian id contains the id of the node at which the guardian was created. Handlers also receive identifiers (*handler ids*). A handler id is unique with respect to its guardian. A handler name is a ⟨guardian_id,handler_id⟩ pair.

Parts of the Argus run-time system are linked into every guardian when it is created. They support various aspects of the Argus programming model such as stable storage and transactions. Other parts of the run-time system are encapsulated in a distinguished guardian called the *guardian manager*. There is a guardian manager at every node of the system. The guardian manager is responsible for creating new guardians, restarting guardians after they crash, and managing resources used by guardians such as releasing stable storage when a guardian is destroyed.

Currently, if a regular guardian does not know a handler's address when it makes a handler call, it generates a lookup request to the guardian manager at its node. The guardian manager can take apart the guardian id part of a handler name and "know" where that guardian (and hence the handler) is located. Currently, guardians do not move, so the guardian manager just returns the given handler name and the node embedded in the guardian id.

2.3 Moving objects in Argus

Object movement can be viewed in two ways. One view is that objects have unique proper names by which they will always be known. When an object moves, it keeps that name and is the same object as before. When it is destroyed, no other object can have its name. This scheme models the world of physical objects. When a physical object is moved, it retains its identity independent of location.

The second view is that objects are replaced. That is, moving an object is really creating a new object in the new place, copying the state of the old object to the new object, and destroying the old object. The old name is now an alias for the new object. These two views are equivalent as far as finding objects is concerned. Either way, the client only has to have a name to find an object. However, the replacement view is more flexible. We can "split" objects. For example, suppose we have an object that has some number of operations. If we wanted to separate the operations into two groups, we could replace the single object with two new objects and alias the old operation names with the respective operations in the new objects. We can also "merge" objects by replacing two or more objects with a single object. Both splitting and merging objects can be implemented completely within the run-time system, causing no changes in clients. On the other hand, to split or merge objects in the proper name view would require that clients be able to handle changes in names.

For the Argus location service, we have chosen the replacement view because of its generality. The computational model of Argus allows (only) guardians and handlers to be replaced. All other objects are encapsulated by guardians so their movement is implied by the replacement of guardians. Handlers are organized by guardian, but may be moved in the sense that a single guardian's handlers can be bound to different replacement guardians' handlers. We assume that the effect of replacement is atomic. A guardian either moves completely or remains at the original node. A guardian cannot be "partially" moved. We also assume that replacements for a particular guardian are infrequent.

We expect that there will be two types of replacement in Argus. The more common type will be straight relocation, where a guardian is replaced by an identical guardian (meaning it provides the same interface) at a different node. The other type is subsystem replacements that may involve replacing multiple guardians and binding their handlers in arbitrary combinations. A summary of the basic replacement method is the following[1]:

1. Start a transaction.
2. Stop the activity at the old guardian(s).
3. Create the new guardian(s).
4. Transfer the state from the old guardian(s) to the new guardian(s). This allows the new guardian(s) to take up where the old one(s) left off.
5. Bind the handlers of the old guardian(s) to those of the new guardians(s).
6. Destroy the old guardian(s).
7. Commit the transaction

If any of these steps fails, then the transaction aborts and the replacement fails. The old guardian(s) recover and resume as if they had crashed. If all of the steps succeed, the transaction commits, and the replacement takes over.

When a guardian is replaced, calls to its handlers need to be routed to its new location. We want the run-time system to do this automatically to support the view that handler invocation is location-independent [13] so that application programmers will not need to

[1]This method is due to Bloom's work on dynamic replacement [1].

be concerned with location. We provide support for this view in the form of a location service. The location service records the handler bindings of step 5. A guardian would generate a lookup request to the location service to get the current binding of the handler if it does not know the handler's location.

A simple implementation of the service might be to use a single guardian as the server with the service operations as handlers. However, this implementation is not resilient to failures. If the server crashes or gets partitioned from the rest of the system then the service is unavailable and the system stops. Therefore, we must replicate the service state to obtain high-availability. The rest of this thesis presents the design and construction of such a service.

Chapter Three

A Highly-Available Location Service

In the next three chapters, we present the design and implementation of a highly-available location service for Argus. Our approach will be to present the basic structures and algorithms in this chapter, ignoring implementation issues like transactions and reconfiguration. Chapter 4 will describe an implementation in the Argus system that addresses the issue of transactions, the probl· ms it causes, and the solutions to those problems. Reconfiguration is addressed in Chapter 5.

For this chapter, we make the following assumptions:

- *all replacement transactions commit*
- the server has a fixed number of replicas in known locations

As discussed in Chapter 2, both guardians and handlers move in Argus. However, the run-time system is really only interested in handler addresses. Moving a guardian also implies moving its handlers, so moving a guardian is an optimization for moving each of its handlers.

Recall that a guardian id contains the id of the node at which the guardian was created. As a result, a handler's name is also its initial address. We write a handler address also as a ⟨guardian_id, handler_id⟩ pair. If a guardian does not know the location of a handler when a call is made, it generates a lookup request to the guardian manager at its node. Since guardians currently do not move, the returned address is the same as the sent one and a call is always made. If the call is to a non-existent guardian, the guardian manager at the called node generates a failure exception that is raised in the calling guardian.

The location service will allow us to move guardians in Argus by keeping track of the new handler addresses. It will also allow us to originate the failure exception locally (via the server) for non-existent guardians instead of waiting until after the call is tried at the called node.

This chapter begins with a presentation of Liskov's multipart timestamp technique, a replication technique for constructing highly-available services. In Section 3.2, we describe the state and operations of the location service and give an example of how the service is used. In Section 3.3, we present an abstract implementation of a replica. Finally, in Section 3.4, we discuss how the clients use the location service in the Argus system.

3.1 Multipart timestamp technique

The multipart timestamp technique was developed as an optimization for the Argus orphan detection algorithm [26]. In the multipart timestamp technique, the data is replicated as in other methods, but updates and reads occur at any (one) replica. This increases the chance that the service will be available and is more efficient than trying to access multiple replicas. To keep the replicas up-to-date, information is propagated in the background as "gossip" messages. While the system is running smoothly, information propagates quickly. However, this is not always the case due to crashes and partitions. A replica may have out-of-date information when a client does a lookup. This is not a problem if the client does not need to know the most up-to-date information, but just something that is "recent enough." The technique is suitable for services where clients do not need to know the most up-to-date information. To improve its utility, the technique gives clients a way of specifying how recent the information must be.

The information in the service can be considered as a set of states in which each state represents the effects of some number of update operations. A state S_1 is more recent than a state S_2 if every update represented in S_2 is also represented in S_1. The service associates a timestamp with a particular state of the information. The timestamps are partially ordered and must meet one invariant: later timestamps are associated with more recent states. Each replica maintains a current state and its timestamp.

There are two types of operations in a service. *Update* operations change the service state and increment the timestamp. They return the new timestamp to clients, thus identifying a state in which the update operation has taken effect. *Query* operations read the service state. They take a timestamp argument; the service guarantees that the answer will come from a state with a timestamp at least as late as the argument timestamp. Thus, if a client needs an answer that reflects a particular update, it can send as a query argument a timestamp known to be greater than or equal to the timestamp returned by that update and be sure that the answer it gets is recent enough.

If the argument timestamp of a query is less than or equal to the timestamp at the replica, the replica can answer the query immediately. Otherwise, the replica has to wait until it has more recent information. Although a client might ask for a "future" state, in practice, clients will only ask for states that exist, so a replica only has to wait until the information needed is propagated to it from the other replicas.

It is necessary that each replica be able to generate a new timestamp independently or else we would be dependent on a timestamp service having the same problems we are trying to solve and will not have gained anything. The notion of the *multipart timestamp* (henceforth, just timestamp) is introduced as a solution. The timestamp is a sequence $\langle t_1, t_2, \ldots, t_k \rangle$ where t_i is the local time (either logical or real) at replica i and k is the total number of replicas in the server. Each part can be incremented independently of any other part, and the ith replica increments the timestamp by advancing only the time in the ith part. Since each replica advances only its own part, the timestamps produced by different replicas are unique and can be generated independently. Among the operations provided by timestamps are the following:

```
merge = proc (ts1,ts2:timestamp) returns (timestamp)

    % Returns a timestamp ts' where ts'[i] = max(ts1[i],ts2[i]), for i = 1,...,k.
    % (ts[i] refers to ts_i)

equal = proc (ts1,ts2:timestamp) returns (bool)

    % If ts1[i] = ts2[i], for i = 1,...,k, then returns true else returns false.

lt = proc (ts1,ts2:timestamp) returns (bool)

    % If ts1[i] ≤ ts2[i], for i = 1,...,k, and ∃ j, such that ts1[j] < ts2[j],
    % then returns true else returns false.
```

If timestamp S is not less than or equal to T and T is not less than or equal to S, then S and T are *incomparable*.

Information is propagated in the background in the form of "gossip" messages. The gossip scheme is based on those used in solutions to the dictionary problem posed by Fischer and Michael [3, 29]. Replicas keep a list (the *send_gossip* list) of update operations that have occurred at that replica and their timestamps and the update operations they receive from the gossip of other replicas. Periodically, a replica sends its *send_gossip* list and its current timestamp to all other replicas in the service.

When a replica receives a gossip message, it compares the timestamp of the message and its timestamp. If the message's timestamp is less than or equal to the replica's timestamp, the message is discarded. Otherwise, each entry in the gossip list is read, and it is determined whether the replica has heard of that operation. If the entry's timestamp is less than the replica's timestamp, the entry is ignored; otherwise, the entry is processed in the same manner as if that update had been invoked at that replica, except that the replica's timestamp is not incremented. When all of the entries in the gossip message have been processed, the message's timestamp is merged into the replica's timestamp.

An important practical consideration for the multipart timestamp technique is how to remove obsolete information from a replica state. For example, the *send_gossip* list will grow without bound if we do not remove any entries. However, we cannot remove an entry from the list at just any time. We have to wait until all of the other replicas know about the operation in question. We do this by keeping a table (the *gossip_table*) of the timestamps the other replicas send with their gossip messages. If the sent timestamp is greater than the timestamp of an entry in the *send_gossip* list, then the sending replica has heard about that update operation. When all of the timestamps in the *gossip_table* are greater than or equal to the entry's timestamp, all of the replicas have heard about the update operation, and the replica can remove that entry from its *send_gossip* list.

Other forms of garbage collection in the replica state will also be necessary. However, the structures in question are specific to the type of service and information provided, so we will defer discussion of these forms until we present the abstract implementation of the location service.

A client uses the service by sending an appropriate message to a replica. The replica responds with a reply message. If the response is slow, the client may send the message to a different replica, or it might send messages to more than one replica in parallel. As stated before, we make no assumptions about the delivery of messages. To make things more efficient, a client can maintain a local cache of the information and timestamps it obtains from the service, and thus avoid some making queries.

3.2 Operations of the location service

In this section, we describe the operations of the location service. First, we present the specifications of the location service. We model the service as an abstract data type, giving its representation and operations. Then we use an example to illustrate the uses of the service operations and the different types of replacements that might happen in the Argus system.

3.2.1 Specifications

The location service provides four operations: enter_guardians, delete_guardian, rebind, and lookup. *Enter_guardians* is called by the guardian manager to enter one or more guardian ids at the server; this operation is intended to be used by the guardian manager to "pre-enter" guardian ids so that entering a guardian id will not have to be done during the actual creation of a guardian. Creation is a fairly slow process, and entering a guardian's id during its creation would make it even slower since the guardian manager would have to wait for the return of the *enter_guardians* operation before allowing the creator operation to return. The operation allows several guardian ids to be entered at once to reduce the number of interactions with the location service. *Delete_guardian* is called by the guardian manager to delete a guardian id from the server. It is called when a guardian manager destroys a guardian independent of a replacement. *Rebind* is called by the replacement system to rebind handlers moved in a replacement transaction and to delete the guardian ids of the replaced guardians. Finally, *lookup* is called by the run-time system to find the address of a handler.

The remainder of this section gives a precise specification of these operations. The specification is given in terms of an abstract model [17] of the location service, which is viewed as consisting of the following components:

ts	A timestamp identifying the current state of the service.
hmap	A record of handler bindings. This is a mapping of handler addresses to handler addresses. A handler h1 is bound to h2 when we want the handler calls to h1 to be routed to h2.
gmap	A record of guardian bindings. This is a mapping of guardian ids to guardian ids. A guardian g1 is bound to g2 when we want all of the handlers in g1 to be bound to the corresponding handlers in g2, except those already bound in *hmap*.
entered	The set of all guardian ids that have ever been entered.
deleted	The set of all guardian ids that have ever been deleted.

A third set, the *exists* set, is the difference between the *entered* and *deleted* sets.

Before we begin the discussion of the service operations, we define some terminology. A guardian id *exists* at the server if it is in the *exists* set. A guardian id is *deleted* at the server if it is on the *deleted* set. The left component of a binding is its *source*. The right component of a binding is its *target*. We also speak of a guardian id as being a source or a target if it appears in the source or the target of a binding, respectively.

The *enter_guardians* operation has the following interface:

```
enter_guardians = proc (gset:SetOfGids) returns (timestamp)
```

The operation takes a set of guardian ids as an argument. The service adds the guardian ids to the *entered* set, increments its timestamp, and writes both to stable storage. Then it returns its timestamp as a result. We require the system to generate unique ids for guardians and not to reuse them. In particular, we require that a guardian id not be entered at the service after being deleted from the service.

The *delete_guardian* operation has the following interface:

```
delete_guardian = proc (g:guardian_id) returns (timestamp)
```

It takes a guardian id as an argument. The service adds the guardian id to the *deleted* set, increments its timestamp, and writes both to stable storage. Then it returns its timestamp as a result.

The *rebind* operation has the following interface:

```
rebind = proc (gm:MapOfGids, hm:MapOfHandlers, t:timestamp)
        returns (timestamp)
```

It takes a guardian id map, a handler address map, and a timestamp as arguments. (The reason for the timestamp argument will be explained later.) The guardian id map is a set of bindings of guardian ids to guardian ids. The handler address map is a set of bindings of individual handler addresses to handler addresses. The service adds the bindings in gm to *gmap*, adds the bindings in hm to *hmap*, adds the source guardian ids to the *deleted* set, increments its timestamp, and writes these structures to stable storage. Then it returns its timestamp as a result. The returned timestamp is guaranteed to be greater than or equal to the argument timestamp.

We can view the information in *exists*, *hmap* and *gmap* as a directed graph where the vertices are handler addresses and the edges are the handler address bindings. The edges are directed from the source to the target of a binding. Guardian ids are conceptually expanded to their corresponding handlers, and guardian id bindings are conceptually expanded to their corresponding handler address bindings for this graph. If there is both a guardian id binding and handler address binding for a particular handler, the handler address binding prevails. Figure 3-1 shows the graph of a particular state of the service. In this example, guardians G and H each have three handlers, guardians F and K have two handlers, and guardians L and M have one handler.

The information has the following representation invariant:

1. The graph has no cycles.
2. Only one edge may emanate from a vertex.

The information must meet this invariant because the lookup procedure traverses this graph to answer queries about the service state. The first part guarantees termination for the lookup procedure. The second part guarantees a deterministic choice.

We rely on the replacement system to obey certain conditions needed to maintain the representation invariant of the binding information. These conditions insure that the inputs to the *rebind* operation are *well-formed*. We state these conditions here:

```
State of the service:

        gmap: G --> H
              L --> M
        hmap: H,h1 --> K,h1
              H,h2 --> K,h2
              H,h3 --> L,h1
        exists: {F,K,M}

Graph of the information:

        ⟨F,h1⟩

        ⟨F,h2⟩

        ⟨G,h1⟩ ----> ⟨H,h1⟩ ----> ⟨K,h1⟩

        ⟨G,h2⟩ ----> ⟨H,h2⟩ ----> ⟨K,h2⟩

        ⟨G,h3⟩ ----> ⟨H,h3⟩ ----> ⟨L,h1⟩ ----> ⟨M,h1⟩
```

Figure 3-1: A service state represented as a graph

1. The source and target guardian ids of bindings in gm and hm exist. (Recall that gm is the guardian id map argument to the *rebind* operation and hm is the handler address map argument.) This prevents possible bindings that would create loops in the graph.
2. A guardian id is not both a source and a target in gm or hm. This prevents explicit loops in the inputs to rebind.
3. A source of a binding in gm is not bound to two different targets in gm, and a source of a binding in hm is not bound to two different targets in hm. This is a uniqueness condition on the information in the inputs to rebind to prevent more than one edge emanating from a vertex in the graph.
4. The source of a binding in gm or hm is not already the source of a binding in gmap or hmap, respectively. This is a uniqueness condition on the relation between the inputs in rebind and the service state to prevent more than one edge emanating from a vertex in the graph.

Recall that our idea of replacement is that we create the replacement guardians, bind the handlers of the old guardians to them, and then destroy the old guardians. Therefore, these conditions can be met easily.

The *rebind* operation needs a timestamp argument because the lookup algorithm will not work if a replica knows about a rebind but not the enter of the target guardian id(s). The lookup would incorrectly state that the handler had been destroyed in such a case. We solve this problem by having the replacement system send a timestamp that is at least as late as the merge of the timestamps returned by the *enter_guardians* operations for all of the target guardian id(s) as an argument. Since the returned timestamp is guaranteed to be greater than or equal to the argument timestamp, it is the timestamp of a state that contains information about both the enter(s) and the rebind.

The guardian id map is a compact way of indicating that all of a guardian's handlers have been bound one-to-one to its replacement guardian, except for the ones already bound in *hm*. We could have simplified matters by having only handler address bindings, but we expect guardian for guardian replacement to be more common than replacements needing to bind arbitrary handlers. Guardian id binding also allows us to replace a guardian without having to know the exact number of handlers that it has.

The *lookup* operation has the following interface:

```
    lookup = proc (h:handler_address, t:timestamp)
           returns (handler_address, timestamp) signals (handler_destroyed)
```

It takes a handler address and a timestamp as arguments. If the handler address argument is bound in the abstract state, it returns a handler address and its current timestamp as results; otherwise, it signals *handler_destroyed*. The returned timestamp is guaranteed to be greater than or equal to the argument timestamp. A handler address is bound in the abstract state if:

- there is a path of length zero or more emanating from the vertex labeled with the handler address argument in the graph of binding information and
- the guardian id at the end of the path is in the *exists* set.

Lookup is a query, so a replica can reply to a lookup only if its timestamp is greater than or equal to the argument timestamp.

3.2.2 An example

To make our discussion more concrete, we present an example of how the location service would be used and what information it keeps. Figure 3-2 goes through this example pictorially. To simplify the presentation we will assume that all of the updates and lookups occur at one replica so that we can ignore the gossip.

Before we begin the example, we define some more terminology. A guardian is *created* when the creator call returns to the creating transaction. A created guardian is assigned

```
(1). F and G are created:

   F       G                                     gmap: <empty>
                                                 hmap: <empty>
   h1      h1                                    exists: {F,G,H,K,L,M,N}
   h2      h2
   h3      h3


(2). After destroying F and  moving G:

   G ----->H                                     gmap: G --> H
                                                 hmap: <empty>
   h1      h1                                    exists: {H,K,L,M,N}
   h2      h2
   h3      h3


(3). After "splitting" H:

   G ----->H        K                            gmap: G --> H
                                                 hmap: H,h1 --> K,h1
   h1      h1 ----->h1                                 H,h2 --> K,h2
   h2      h2 ----->h2                                 H,h3 --> L,h1
   h3      h3 --                                 exists: {K,L,M,N}
               |    L
               |
               ---->h1


(4). After moving L:

   G ----->H        K                            gmap: G --> H
                                                       L --> M
   h1      h1 ----->h1                           hmap: H,h1 --> K,h1
   h2      h2 ----->h2                                 H,h2 --> K,h2
   h3      h3 --                                       H,h3 --> L,h1
               |    L ------>M                   exists: {K,M,N}
               |
               ---->h1       h1


(5). After "merging" K and M:

   G ----->H        K                  N         gmap: G --> H
                                                       L --> M
   h1      h1 ----->h1 --------------->h1        hmap: H,h1 --> K,h1
   h2      h2 ----->h2 --------------->h2              H,h2 --> K,h2
   h3      h3 --                  ----->h3             H,h3 --> L,h1
               |    L ------>M   |                     K,h1 --> N,h1
               |                 |                     K,h2 --> N,h2
               ---->h1       h1 --                     M,h1 --> N,h3
                                                 exists: {N}
```

Figure 3-2: An example of using the location service

a guardian id that exists by the guardian manager. A guardian is *destroyed* when the destroying transaction commits. A destroyed guardian's id is deleted from the service by the guardian manager.

We begin the example by invoking:

    enter_guardians ({F,G,H,K,L,M,N})

This enters the guardian ids F, G, H, K, L, M, and N into the *exists* set. Next, we create two guardians that are given guardian ids F and G; each has three handlers. Step (1) shows the replica state after the ids are entered and guardians F and G are created. Both maps are empty and there are seven guardian ids in the *exists* set.

Continuing the example, we destroy guardian F, resulting in the call:

    delete_guardian (F)

Next, we move G. We do this by creating a guardian, also with three handlers, that is given the guardian id H. We bind G to H by invoking:

    rebind ({⟨G,H⟩},{ },ts)

The resulting state is shown in step (2). F and G have been destroyed. There is an entry in *gmap*, and F and G are no longer in the *exists* set. A lookup of ⟨F,h1⟩ would result in a *handler_destroyed* exception because there are no bindings for F in either map and F is not in the *exists* set. A lookup of ⟨G,h1⟩ would first find a path from G ending at H. Since H is in the *exists* set, the operation would return ⟨H,h1⟩.

In step (3), we "split" H. We do this by creating guardian K with two handlers: h1, h2; and guardian L with one handler, h1. We bind H's handlers by invoking:

    rebind ({ },{⟨⟨H,h1⟩,⟨K,h1⟩⟩,⟨⟨H,h2⟩,⟨K,h2⟩⟩,⟨⟨H,h3⟩,⟨L,h1⟩⟩},ts)

Then H is destroyed. Now when a lookup operation of ⟨G,h1⟩ is done, the service follows the path in *gmap* to H and then sees that ⟨H,h1⟩ has been bound to ⟨K,h1⟩ in *hmap*, which is the end of the path. K is in the *exists* set, so it returns ⟨K,h1⟩. When a lookup of ⟨G,h3⟩ is done, the service gets to H again, but sees that ⟨H,h3⟩ is bound to ⟨L,h1⟩, so ⟨L,h1⟩ is returned.

Next, we move L. We create guardian M with one handler, h1. We invoke:

    rebind ({⟨L,M⟩},{ },ts)

Then L is destroyed. Step (4) shows the state after this is done. A lookup of ⟨G,h1⟩ still returns ⟨K,h1⟩ (K is still in the *exists* set), but now a lookup of ⟨G,h3⟩ returns ⟨M,h1⟩, since L is bound to M in *gmap* and M is in the *exists* set.

Finally, we "merge" K and M together. We create guardian N with three handlers: h1, h2, h3. We invoke:

rebind ({ },{⟨⟨K,h1⟩,⟨N,h1⟩⟩,⟨⟨K,h2⟩,⟨N,h2⟩⟩,⟨⟨M,h1⟩,⟨N,h3⟩⟩},ts)

Then K and M are destroyed. This final state is shown in step (5). A lookup of ⟨G,h1⟩ returns ⟨N,h1⟩, and a lookup of ⟨G,h3⟩ returns ⟨N,h3⟩.

3.3 Abstract implementation

In this section, we present an abstract implementation of our location service. It is abstract in the sense that we only deal with abstract data types and operations. We ignore issues like transactions and reconfiguration. We deal with these issues in later chapters. For the first four parts of this section, we also assume that replicas do not crash. The last part of this section discusses how to deal with replica crashes through the use of stable storage.

We assume that replica processes access data structures one at a time. That is, each of the processes runs in a critical section. This is easy to implement in Argus using the mutex construct.

3.3.1 Data structures

A replica is implemented by a guardian. We begin by describing the state of a replica. The state of the replica has many components. We will use the following notation to describe the types of the various components:

{item_type}	denotes a set of items of type item_type
⟨item1, item2⟩	denotes an ordered pair where the first component is of type item1, and the second component is of type item2.
key → data	denotes a function mapping items of type key to items of type data. An individual ⟨key, data⟩ pair is referred to as an *association.*

Gmap is the map of guardian id to guardian id bindings. Its type is:

MapOfGids = guardian_id → guardian_id;

Hmap is the map of handler address to handler address bindings. A handler address is a ⟨guardian_id, handler_id⟩ pair. The type of *hmap* is:

MapOfHandlers = ⟨guardian_id, handler_id⟩ → ⟨guardian_id, handler_id⟩;

Exists is a set of guardian ids. It is the set of currently existing guardians; it is the same as the *exists* set described in Section 3.2. Its type is:

SetOfGids = { guardian_id };

Deleted is a set of ⟨guardian_id, timestamp⟩ pairs. It is a partial set of the guardian ids that have been deleted along with the timestamp of the *delete_guardian* or *rebind* operation associated with the delete event. Its type is:

```
SetOfDeletedGids = { ⟨guardian_id, timestamp⟩ };
```

Send_gossip is a set of ⟨timestamp, update_record⟩ pairs. It is a partial set of the update operations reflected in the replica's current state along with the timestamp of the operation. Its type is:

```
SetOfUpdate_records = { ⟨timestamp,update_record⟩ }
```

An update_record represents an update operation. Its type is a discriminated union:

```
update_record = oneof [enter   : SetOfGids,
                       rebind  : rebind_entry,
                       delete  : guardian_id]

rebind_entry = struct[gm:MapOfGids, hm:MapOfHandlers]
```

The field components of the arms of the update record are the arguments to the operation represented.

Gossip_table associates timestamps to replicas and is an array of timestamps. *Ts* is the current timestamp of the replica. *My_index* is the index of the replica in *ts* and *gossip_table*. *My_part* is the value of the replica's part in *ts* (that is, *ts[my_index]*). Figure 3-3 summarizes the state of a replica. We will use the notation *variable.component_name* to refer to the various components of any record-like type. The components of the server state will be referred to by S.*component_name*.

```
gmap          : MapOfGids;
hmap          : MapOfHandlers;
exists        : SetOfGids;
deleted       : SetOfDeletedGids;
send_gossip   : SetOfUpdate_records;
gossip_table  : array[timestamp];
ts            : timestamp;
my_index      : int;
my_part       : int;
```

Figure 3-3: State of a service replica

3.3.2 Operations

For the discussion of the server operations, we define the following procedures on maps:

```
key_is_in = proc (m:map, k:key) returns (bool)

    % Returns true if there is data associated with k in m;
    % otherwise returns false.

fetch_data = proc (m:map, k:key) returns (data)

    % Requires that k be associated with some data in m.
    % Returns the data associated with k in m.

add_association = proc (m:map, a:assoc)

    % Adds a to m.  It overwrites any previous association to a.key in m.

remove_association = proc (m:map, k:key)

    % Removes the association ⟨k,?⟩ from m if there is one.
```

We also define the procedure:

```
wait_until = proc (condition:bool)

    % The process executing this procedure waits for the condition to be
    % true before proceeding.
```

Recall that a client can call the same operation with the same arguments at multiple replicas (if the initial call is too slow to respond or they were done concurrently). This can lead to duplicate operation messages. A duplicate *enter_guardians* message may cause problems if it is "late", causing a replica to reenter a guardian id after it has been deleted. We implement the *enter_guardians* operation without checks for duplicate messages, because the concrete implementation in the next chapter prevents duplicate operations. However, some of the alternative solutions proposed in the next chapter do not prevent duplicate operations. For implementations of these solutions, we would first check if a guardian id is in the *deleted* set before putting it in the *exists* set. Duplicate *delete_guardian* and *rebind* messages cause no harm because the reintroduction of the information they carry only causes extra work for the garbage collection procedures (although we would probably want to check for them in a real implementation to avoid wasted effort).

The service operations are implemented in the following manner:

```
enter_guardians = proc (gset:SetOfGids) returns (timestamp)

    S.my_part := S.my_part + 1;
    S.ts[S.my_index] := S.my_part; % advance my part in ts as well
    S.exists := S.exists ∪ gset
    S.send_gossip := S.send_gossip ∪ {⟨S.ts,make_enter(gset)⟩};
    S.gossip_table[S.my_index] := S.ts
    return (S.ts);

    end enter_guardian


delete_guardian = proc (g:guardian_id) returns (timestamp)

    S.my_part := S.my_part + 1;
    S.ts[S.my_index] := S.my_part;
    S.exists := S.exists - {g};
    S.deleted := S.deleted ∪ {⟨g,S.ts⟩};
    S.send_gossip := S.send_gossip ∪ {⟨S.ts,make_delete(g)⟩};
    S.gossip_table[S.my_index] := S.ts
    return (S.ts);

    end delete_guardian


rebind = proc (gm:MapOfGids,hm:MapOfHandlers,t:timestamp) returns (timestamp)

    % "?" matches any

    wait_until (S.ts ≥ t)
    sources : SetOfGids := {g | ∃ ⟨g,?⟩ ∈ gm ∨ ∃ ⟨⟨g,?⟩,⟨?,?⟩⟩ ∈ hm}

    S.my_part := S.my_part + 1;
    S.ts[S.my_index] := S.my_part;
    S.exists := S.exists - sources;
    for g ∈ sources do S.deleted := S.deleted ∪ {⟨g,S.ts⟩} end;
    for g_assoc ∈ gm do add_association (S.gmap, g_assoc) end;
    for h_assoc ∈ hm do add_association (S.hmap, h_assoc) end;
    S.send_gossip := S.send_gossip ∪ {⟨S.ts,make_rebind(
        make_rebind_entry (gm,hm))⟩};
    S.gossip_table[S.my_index] := S.ts;
    return (S.ts);

    end rebind
```

```
lookup = proc (g:guardian_id,h:handler_id,t:timestamp)
        returns (guardian_id, handler_id, timestamp)
        signals (handler_destroyed)

    wait_until (S.ts ≥ t)
    while true do
        if key_is_in (S.hmap, ⟨g,h⟩)then
                ⟨g,h⟩ := fetch_data (S.hmap,⟨g,h⟩);
            elseif key_is_in (S.gmap, g) then
                g := fetch_data (S.gmap, g);
            elseif g ∈ S.exists then
                return (g,h,S.ts);
            else
                signal (handler_destroyed);
            end;
        end; % while

    end lookup
```

Guaranteeing that a replica will have a timestamp greater than or equal to an argument timestamp means that a replica may have to wait to do an operation if its timestamp is not large enough. This is not a problem for lookups since they do not tie up replicas. However, rebinds tie up replicas, so we do not want to have them wait long. Since typically guardian ids are pre-entered long before they are used, we expect that all replicas will know about the enters and have timestamps that are large enough by the time a *rebind* operation is invoked, so there will be no wait. An alternate strategy would have been to make the *rebind* operation responsible for adding the target guardian id(s) to the *exists* set as well. Then the operation would not need a timestamp argument. However, we would still have to have the *enter_guardians* operation to pre-enter guardian ids for normal guardian creation.

Recall that we assume that the replacement system follows the well-formedness conditions for the inputs to the *rebind* operation. We did not explicitly check for these conditions here, but in a real implementation, it would be a good idea to do so, to guard against replacement system failures.

3.3.3 Gossip

A replica periodically sends a gossip message to all other replicas. How often a replica sends gossip depends on two factors: how often and for how long we expect a replica to crash, and how expensive it is to send messages. If we expect that replicas will crash often or for long periods of time, we would want to gossip as soon as possible after the replica performs an update. This is because if a replica goes down before getting a chance to gossip, all queries that need to have that update will be delayed until the replica comes back up. This implies smaller messages being sent often. On the other hand, if we expect that replicas will not crash often or for very long, we can accumulate update operations and send them in one bunch. The messages are larger, but are sent less often. We think the latter will be the case most of the time.

A gossip message M has components *gossip_list*, which is the *send_gossip* list of the sending replica, *ts*, the current timestamp of the sending replica, and *index*, the index of the sending replica in the gossip table. Upon receiving a gossip message, the receiving replica invokes the following procedure:

```
gossip_processing = proc ( )

    S.gossip_table[M.index] := merge (S.gossip_table[M.index], M.ts)
    if M.ts ≤ S.ts then return end;
    for u ∈ M.gossip_list do
        if u.ts ~≤ S.ts then
            tagcase u.rec of
                enter (gset:SetOfGids) :
                    S.exists := S.exists ∪ gset -
                        {g | g ∈ gset ∧ ∃ d ∈ S.deleted s.t. g = d.guardian_id};
                        % some of the guardian ids may have already been deleted
                delete (g:guardian_id) :
                    S.exists := S.exists - {g};
                    S.deleted := S.deleted ∪ {⟨g,u.ts⟩};
                rebind (r:rebind_entry) :
                    sources : SetOfGids := {g | ∃ ⟨g,?⟩ ∈ r.gm ∨ ∃ ⟨⟨g,?⟩,⟨?,?⟩⟩ ∈ r.hm}
                    S.exists := S.exists - sources;
                    for g ∈ sources do S.deleted := S.deleted ∪ {⟨g,u.ts⟩} end;
                    for g_assoc ∈ r.gm do add_association (S.gmap, g_assoc) end;
                    for h_assoc ∈ r.hm do add_association (S.hmap, h_assoc) end;
                end; % tagcase
            S.send_gossip := S.send_gossip ∪ {u};
            end; % if
        end; % for
    S.ts := merge (S.ts, M.ts)
    S.gossip_table[S.my_index] := S.ts

    end gossip_processing
```

In this gossip scheme, a replica sends its entire *send_gossip* list to all other replicas regardless of what the *gossip_table* says the other replicas already know. The garbage collection algorithm removes an entry only when a replica knows that it is known by all replicas. If a replica that does not know (or that the sender thinks does not know) about an entry goes down or is partitioned from the sender for a long period of time, that entry may be sent many times to other replicas that already know about it. An approach for reducing the sizes of gossip messages is to not send an entry to replicas that know about it already. This would reduce the size of gossip messages and the amount of busy processing at the receiving replica, but it does so at the expense of space or computational time at the sending replica. A replica would either keep separate *send_gossip* lists for each replica, or it would generate a new list for each replica every time it gossips.

3.3.4 Garbage collection

Garbage collection of the *send_gossip* list is straightforward. It is implemented by the following procedure:

```
gossip_cleanup = proc ( )

    for u ∈ S.send_gossip do
        if ∀ i, S.gossip_table[i] ≥ u.ts
            then S.send_gossip := S.send_gossip - {u} end;
        end; % for

    end gossip_cleanup
```

Garbage collection also needs to be done on the *deleted* set. Like the case of the *send_gossip* list, the entries in the *deleted* set need to be kept until all the other replicas know about the delete. This is because a gossip message containing the *enter_guardians* operation for a deleted guardian id might arrive after the *delete* operation for that id. If a replica waits until all the replicas know about the delete, then it knows that all future gossip will either contain the *delete* operation or no information for that guardian id. (This is because of our requirement that guardian ids not be entered after they are deleted.)

There is a second problem with garbage collecting the *deleted* set. If there can be duplicate *enter_guardians* messages (from unsuccessful calls), one may arrive after an entry in the *deleted* set has been garbage collected. We can handle this problem by making assumptions about our communications network. We assume that the clocks at all nodes are approximately synchronized with maximum skew of ε [11, 20], and we impose an upper bound δ on message delay. Messages are marked with the real time at the sending node. At the receiving replica, messages that are marked as more than $\varepsilon + \delta$ older than the local time are discarded. (That is, we pretend such a message never arrived.) It is not difficult to choose reasonable values of ε and δ -- each can be quite large -- so this scheme is practical. To handle late *enter_guardians* messages that are not discarded but arrive after the *delete_guardian* message, we retain information about deletes at least $\varepsilon + \delta$ more than the time in the *delete_guardian* message. The *deleted* set becomes a set of ⟨guardian_id, timestamp, time⟩ triples and entries can be garbage collected only if both the timestamp condition and the time condition are met.

Again, we will not give an implementation that takes the late *enter_guardians* messages into consideration because the concrete implementation described in the next chapter eliminates this problem in a different manner. Hence, the algorithm for garbage collecting the *deleted* set is the same as for the *send_gossip* list:

```
deleted_cleanup = proc ( )

    for d ∈ S.deleted do
       if ∀ i, S.gossip_table[i] ≥ d.ts
          then S.deleted := S.deleted - {d} end;
       end; % for

    end deleted_cleanup
```

Garbage collection for *gmap* and *hmap* is trickier since the paths run through both maps. We will call a path that ends with a guardian id in the *exists* set an *active* path. Conversely, a path that ends with a guardian id not in the *exists* set is an *inactive* path. We cannot remove a binding if it is part of an active path. But instead of checking whether a binding is part of an active path, we can instead just check if a given binding is the last binding of an inactive path and remove it if it is. Eventually, all of the bindings in an inactive path are garbage collected since the guardian ids of interior vertices of a path are not in the *exists* set. (This is because the the *rebind* operation deletes the source guardian ids.) The following code implements garbage collection for *gmap* and *hmap*:

```
map_cleanup = proc ( )

    for ⟨g1,g2⟩ ∈ S.gmap do
       if g2 ∉ S.exists then
          if not key_is_in (S.gmap,g2) ∧ not key_is_in (S.hmap,⟨g2,?⟩)
             then remove_association (S.gmap, g1) end;
          end;
       end; % for
    for ⟨⟨g1,h1⟩,⟨g2,h2⟩⟩ ∈ S.hmap do
       if g2 ∉ S.exists then
          if not key_is_in (S.gmap,g2) ∧ not key_is_in (S.hmap,⟨g2,h2⟩)
             then remove_association (S.hmap,⟨g1,h1⟩) end;
          end;
       end; % for

    end map_cleanup
```

3.3.5 Replica recovery

Throughout our discussion, we have been assuming that replicas do not crash to simplify the presentation. Of course, this is unreasonable. When replicas crash, we must concern ourselves with potential information loss. A replica must retain all of the information it has stated that it has in the presence of crashes. In this section, we discuss replica recovery and prevention of information loss.

As explained in Chapter 2, guardians can have a stable state that is stored on stable storage. We can use stable storage to prevent information loss since once information is written to stable storage, it will not be lost with arbitrarily high probability [12]. However, stable storage is slow and expensive; any change made to an object causes the entire object to be rewritten at the commit of the transaction that changed it. We might like to minimize the amount of information kept on stable storage. At the very least, we must keep the replica timestamp and the value of its part on stable storage. A replica keeps the value of its part on stable storage to be able to generate new timestamps after a crash. The replica timestamp needs to be stable so that the replica knows what information it has gossiped to other replicas. Since the garbage collection algorithm for the *send_gossip* list and the *deleted* set rely on replicas not losing information after they say they have the information, a replica would have to write out its timestamp each time it sends gossip.

The rest of the information need not be kept on stable storage provided that the probability of a replica being able to gossip the information before it crashes is high enough. For example, we might gossip to other replicas and wait for acknowledgments before returning to the client. When a replica recovers from a crash, it must ask another replica for a state. Note that this implies that replicas provide a *get_state* operation.

This scheme is complicated. If the transaction aborts, we must be able to undo changes at replicas that received the gossip. We lose availability and efficiency because the update operation must wait for these acknowledgments. Crash recovery is slow since a replica cannot recover without communicating with other replicas. There is also the possibility of the only replica in a partition crashing and not being able to communicate with another replica for a long period of time, effectively stopping the system in that partition.

It is not clear that these complications are worth the trouble of avoiding storing information on stable storage. Although stable storage is slow and expensive, its cost is not that great. It can be argued that the writing of the replica state happens for transactions that do not need to be particularly fast. *Enter_guardians* operations are done infrequently, and a guardian manager would call this operation before it had assigned all of the guardian ids from the previous *enter_guardians* operation. The *delete_guardian* operation can be called any time after the destroying transaction commits. *Rebind* operations are run as part of replacement transactions, and these transactions do not have to be particularly fast since they are rare. It is also very important that the service does not lose information. Therefore, we will keep the entire replica state on stable storage.

The simplest scheme is to put the replica state as described on stable storage and rewrite it each time it is modified. Recovery would consist of reinitializing the volatile variables of the replica guardian. If we want to reduce the amount of information written to stable storage at the commit of an update transaction, we can use the approach presented by Weihl and Liskov [28] for the amap type. The idea is to keep recent changes in a log, which is written out after every update. Periodically, when the log becomes large enough, the replica writes out the replica state with the changes from the log and empties the log to reduce the cost of recovery. In addition, the data can be partitioned into several sets so that only part of the state must be written for any given operation.

3.4 Clients of the location service

There are three types of clients of the location service in Argus. They are the guardian manager, the replacement system, and regular guardians. This section explains how these clients interact with the service, and what information each client must keep.

3.4.1 Guardian manager

The guardian manager keeps a stable timestamp reflecting all its interactions with the location service. Whenever a guardian manager receives a new timestamp, it merges the new timestamp with its timestamp and writes the result to stable storage.

Periodically, the guardian manager enters a set of guardian ids at the server with the *enter_guardians* operation. This set of entered guardian ids is kept on stable storage. When a guardian is created, it is assigned an unused id from this set. Then the id and the guardian manager's timestamp are given to the created guardian. The id and the guardian manager's timestamp are recorded with the other stable information that the guardian manager keeps about its guardians when the creating transaction commits.

If a transaction that creates a guardian aborts, the new guardian is destroyed. (This includes replacement transactions.) In this case, the guardian manager at the created guardian's node must inform the server of the destruction by calling the *delete_guardian* operation.

There is a possible problem if the guardian manager crashes before a transaction creating a guardian commits. Such a crash would cause the transaction to abort, destroying the created guardian. The guardian manager will delete the guardian's id from the service when it recovers, but it needs to determine which id it should delete.

We can handle this in several ways. One is to have the guardian manager delete all ids except the ones belonging to guardians that are currently residing at the node. If we are concerned with throwing away large numbers of unused guardian ids (if the guardian manager enters hundreds of guardian ids at a time), then the guardian manager can keep the last guardian id it assigned on stable storage and only delete the guardian ids that do not belong to current guardians and are less than the saved id. However, this solution is undesirable because it would cause a wait for the write to stable storage during the creation process. A compromise between the two would be to have the guardian manager periodically write out a "high water" mark to stable storage indicating that all of the ids above the mark have not been assigned. Then when the guardian manager recovers from a crash, it deletes the ids below the mark that do not belong to current guardians. This may delete some unused guardian ids, but will not delete all of them.

After a guardian is destroyed, the guardian manager calls *delete_guardian* with the destroyed guardian's id as the argument. When a replacement is done, the guardian manager is asked to destroy the replaced guardians and is given the timestamp of the *rebind* operation by the replacement system.

When a handler call arrives at a node, the Argus system routes it to the proper guardian. If the called guardian is non-existent, the guardian manager returns its timestamp in a signal to the calling guardian that indicates that the handler no longer exists. This timestamp is guaranteed to be at least as late as the rebinding (or destruction) of that handler.

3.4.2 Replacement system

Replacements are done by a (logically) separate replacement system, envisioned to be along the lines of the one proposed by Bloom [1]. The replacement system keeps track of the creation, bindings, and destruction of guardians involved in a replacement. After the replacement guardians have been created, it calls the *rebind* operation to bind the handlers of the replaced guardian(s) to the handlers of the replacement guardian(s) in the server. The replacement system sends the timestamp returned by the the *rebind* operation and the set of guardians to be destroyed to the guardian managers of the source guardians.

3.4.3 Regular guardians

Regular guardians may interact with the location service whenever they make a handler call. Each guardian keeps a stable timestamp that is written to stable storage whenever a transaction commits. Initially, this is its creation timestamp (the timestamp given to it by its guardian manager when it was created). Guardians also keep a cache of lookup results. When a guardian makes a handler call, it looks for the address in its cache first. There are four possibilities:

1. The cache has an entry for the handler address. The call is made, and it is successful. (The cache was up-to-date.)
2. The cache has an entry for the handler address. The call is made, but is not successful. The timestamp returned by guardian manager at the called node is merged into the guardian's timestamp, and a lookup request with the new timestamp is made to the location server.
3. The cache does not have an entry. A lookup request is made with the guardian's timestamp to the location server.
4. The cache has an entry, but there is no response from the called guardian due to a crash or a partition. In this case, it is possible that the called guardian may have moved, so it may be worthwhile for the calling guardian to do a lookup request with its own timestamp to the server. However, we do not guarantee any useful information since the calling guardian's timestamp has not changed and may not be late enough.

For possibilities 2, 3 and 4, the lookup request may return a new address, which is then stored in the cache and used to make another call, or it may signal that the handler has been destroyed. For possibility 4, it may also return the same address.

When a lookup is done, we must pass a timestamp at least a large as the *enter_guardians* timestamp for that guardian id to the location service. We ensure this in the following ways:

1. Guardians send their timestamps in all the messages they send.
2. When a message is received, the message's timestamp is merged into the receiver's timestamp.
3. When a transaction commits, a guardian's timestamp is written to stable storage as part of the information stored by the commit protocol.

These steps ensure that if a guardian id is sent in a message, so is a timestamp greater than or equal to its enter timestamp, so when a lookup occurs, the replica will have (or wait for) the correct information.

A frequent pattern of use in Argus is for the transaction that creates a guardian to then call one of its handlers. The current scheme would require a lookup for this handler's address even though it cannot have moved. A guardian can avoid this lookup by making use of the fact that host names are embedded in guardian ids. It can use a handler name like a cache. If the real cache is empty, the guardian can automatically try at the address the handler name represents. This approach will cause an extra exchange of messages when accessing a moved guardian for the first time after a cache reset (for example, after a crash), but if most guardians do not move, it will speed up lookups in general.

3.4.4 Discussion

The current Argus implementation has a two-level view of handler mapping. Regular guardians maintain a cache that maps handler addresses to handler addresses. Invalidating cache entries is done automatically through the use of a system error code. When a cache entry is missing or invalidated, the guardian makes a lookup request to the guardian manager. (As noted in Chapter 2, the guardian manager can "take apart" the guardian id part of the handler address and "knows" where the host is.) This structure was chosen for modularity so that when the location server was implemented, it could be integrated with fewer changes to the run-time system. It was also done so there could be another cache of information at the guardian manager that might reduce the number of lookups to the server. It is not clear whether a reduction will occur, however. If different guardians rarely invoke common handlers, then the lookup requests will be faster if the guardian does them directly, since it is unlikely that the guardian manager's cache will be any more up-to-date.

A better caching strategy might be to have the guardian manager cache information about rebinds. When a handler call to a non-existent handler is received, the guardian manager returns information in its cache along with its timestamp. The meaning of this would be: "the guardian has been replaced; here is where the guardian went when it happened." This information would be regarded as a hint by the calling guardian, which can decide to use it or generate a new lookup request. Since the information is only a hint, it does not have to be stable and will be lost if the guardian manager crashes. The big savings would be if guardians do not move often or very fast. Then the information would most likely be correct when a handler call arrives.

Since the mechanism was already in place, we attempted to integrate the location server into the current system with as few changes to the run-time system as possible. We continue to have regular guardians send lookup requests to the guardian manager. The guardian manager now does a lookup query to the server instead of taking apart the guardian id and relays any *handler_destroyed* signals. Otherwise, the interaction is as described. We will ignore caches for the rest of the thesis.

The replicated nature of calls to the server is hidden in the server cluster. It provides the interface presented. It is used by the guardian manager and the replacement system. The server cluster manages the policy for calling the various server replicas, such as whether to call many replicas concurrently or when to timeout an operation.

3.5 Summary

In this chapter, we presented the design of a highly-available location service for Argus. We described our choice of replication technique, Liskov's multipart timestamp technique. The operations of the location service were defined, and we gave an example of how the service is used. An abstract implementation was presented for the basic structures and algorithms. We ended the chapter with a discussion of the integration of the location service into the Argus system.

An implementation goal for the location service is that clients be able to make progress as they use the service. If a client does a lookup on a handler name after getting a non-existent handler exception, then it should get a later address than the one it has for that handler name or a *handler_destroyed* exception from the service. This happens in our system because guardians and guardian managers keep track of the events they have seen (by merging the timestamps of these events). Since the non-existent handler exception returns a timestamp at least as late as the rebind or destruction of that handler, a client can ask for an answer to come from a state that reflects the rebind or destruction of the handler. Thus, the client will make progress in finding out the location or destruction of a handler. Even if the answer to the next lookup request is not the latest address and the call signals another non-existent handler exception, the returned timestamp from this exception is guaranteed to be large enough to allow the client to find out the next step.

We made some assumptions to simplify the presentation in this chapter. We have not talked about the interaction of the location server with the transaction system. This issue is addressed in the next chapter. We also assumed a fixed configuration for the server. Chapter 5 looks at reconfiguring the server state.

Chapter Four

Implementing the Location Service in Argus

In this chapter, we relax one of the assumptions made in Chapter 3. We allow the possibility of replacement transactions aborting and running concurrently at the same replica. Aborted transactions and concurrency add complexity to our location service. Since our implementation calls the *rebind* operation as part of a replacement transaction, this leads to problems for our server. Consider the following scenario (shown pictorially in Figure 4-1):

We have a service with three replicas, although for simplicity we will assume that all operations occur at replica 1. The current state at each of the replicas is the same and the current timestamp is ⟨1,1,1⟩. Guardian ids A, B, C, and D have been entered, and guardians A and C have been created. Part (1) of the figure shows this initial state at replica 1.

A replacement transaction *P* moves A. It creates guardian B and invokes:

rebind ({⟨A,B⟩}, { }, ⟨1,0,0⟩)

The new state of replica 1 is shown in part (2) of the figure. The operation returns the timestamp ⟨2,1,1⟩.

A guardian K does a lookup of ⟨A,h1⟩ with timestamp ⟨0,0,0⟩ and receives (B,h1,⟨2,1,1⟩).

Now suppose *P* aborts. This means that A did not really move. The state shown in part (2) is not correct, and guardian K got the wrong answer. We have to be able to undo the effects of *P*, and should not give out information about changes made by uncommitted transactions. So in this case, when K does the lookup the service should answer with (A,h1,⟨1,1,1⟩) and when *P* aborts, the state of replica 1 will return to part (1).

Now suppose instead that *P* is still in progress (neither aborted nor committed, we indicate this in the figure by a "*"), and a replacement transaction *Q* moves C. It creates guardian D and invokes:

rebind ({⟨C,D⟩}, { }, ⟨1,0,0⟩)

The operation returns timestamp ⟨3,1,1⟩. Then transaction *Q* commits, so its changes are allowed to be given out. Part (3) shows this situation.

```
(1). Initial situation:

    A                   C                           gmap: <empty>
                                                    hmap: <empty>
    h1                  h1                          exists: {A,B,C,D}
    :                   :                           ts: <1,1,1>
    :                   :


(2). After B is rebound:

    A -----> B          C                           gmap: A --> B
                                                    hmap: <empty>
    h1       h1         h1                          exists: {B,C,D}
    :        :          :                           ts: <2,1,1>
    :        :          :

(3). After C is rebound:

    A -----> B          C -----> D                  gmap: A --> B*
                                                          C --> D
    h1       h1         h1       h1                 hmap: <empty>
    :        :          :        :                  exists: {B,D}
    :        :          :        :                  ts: <3,1,1>
```

Figure 4-1: Transaction scenario

A guardian *G* with timestamp ⟨0,0,1⟩ wants to call the first handler of C, so it invokes:

lookup (C,h1,⟨0,0,1⟩)

This returns (D,h1,⟨3,1,1⟩) and G's timestamp becomes ⟨3,1,1⟩. Later, G wants to call the first handler of A, so it invokes:

lookup (A,h1,⟨3,1,1⟩)

This call should be answered since the replica's timestamp is equal to the argument timestamp, but it cannot be answered. Since *P* is still in progress, its changes are not available. We cannot answer the request with (A,h1,⟨1,1,1⟩) because if transaction *P* commits, the information would come from a state that is not late enough. We cannot answer (B,h1,⟨3,1,1⟩) because if *P* aborts, it would be incorrect.

The problem is that the scenario violates the invariant that larger timestamps are associated with more recent states. Since state changes happen at the commits of transactions and the timestamps generated at a single replica are comparable, update operations done at the same replica should be assigned timestamps in the order they commit. That is, the state reflecting *Q*'s change should have an earlier timestamp than the state reflecting *P*'s change. Then a transaction that finds out about *Q*'s change will still get a timestamp less than the state reflecting *P*'s change.

There are several ways to solve these problems. In this chapter, we present a complete implementation of one of the solutions called the *serial* solution. Then we consider some alternative solutions and how they address various issues. Finally, we state our conclusions about the serial solution and compare it to the alternative solutions.

4.1 Serial solution

To know when a transaction's changes must be undone due to an abort, the server must interact with or be part of the transaction system. The most straightforward way for the server to find out the outcome of a transaction is through the use of atomic objects. In our implementation, we make the service state atomic to provide the synchronization and recoverability needed to handle aborted transactions.

However, making the entire service state atomic is overkill. Since an update operation acquires a write lock on the state, it would exclude any other operation from accessing the state. The lock could be held for a potentially long time since it is not released until the update transaction commits. This could delay other operations from accessing the state. In particular, it would delay lookup operations. This would be undesirable because lookups are expected to be done frequently. In addition, modifications to the replica state done by other "operations" (that is, gossip processing and garbage collection) never need to be undone, so making the entire state atomic would add unnecessary complexity and overhead to these operations. Our implementation separates these concerns by having update operations write an update record to an atomic log. After the update transaction commits, a background process reads the update record from the log and does the actual changes to the replica state. This way, other operations can read and change the replica state while an update operation is in progress.

Thus, we allow updates to run concurrently with other operations. However, concurrent updates are still a problem. They can still commit "out of order." One way to solve this problem is to restrict updates to preclude the problem. In the serial solution, we do this by making updates serial at each replica. That is, we allow only one update operation to run at a replica at any given time until it commits or aborts. This means that operations occurring at the same replica will always commit in the "correct" order since the earlier operation must commit before the later one starts.

Now we continue our implementation from Chapter 3, adding the components and code necessary to handle transactions. First, we describe the new data structures in the replica state. Then we discuss the changes in replica processing.

4.1.1 Data structures

We add the following data structures to the state:

In_progress is an atomic log. The log entries are ⟨timestamp,update_record⟩ pairs. The log supports the following operations:

```
    add_entry = proc (l:log, e:log_entry)

        % Adds e to the end of l.

    remove_entry = proc (l:log) returns (log_entry)

        % If the log is not empty, it removes the first entry from l and returns it.

    empty = proc (l:log) returns (bool)

        % Returns true if the log is empty; false if it is not.
```

The log is implemented by a user-defined atomic type with similar semantics to the semi-queue type described by Weihl and Liskov [28]. This type was chosen to allow processing of committed updates to be done concurrently while another update is in progress. We can add entries to the log at any time. We can remove any entry from the log as long as the transaction that added it has committed. Aborted transactions result in log entries that are ignored by the remove_entry procedure. The use of a built-in atomic type would have caused remove_entry operations and add_entry operations to conflict since both modify the state of the log.

Lock is the replica lock used to restrict updates to be serial. It is implemented by an atomic_record of one null object. We chose the atomic_record data type because it supports a test_and_write operation that allows us to test whether a process can get a write lock and obtain the lock (if possible) in one indivisible step. A write lock blocks out attempts by other transactions to obtain the lock and is also held until the transaction finishes.

Figure 4-2 shows the state of an implemented replica.

```
gmap          : MapOfGids;
hmap          : MapOfHandlers;
exists        : SetOfGids;
deleted       : SetOfDeletedGids;
send_gossip   : SetOfUpdate_records;
gossip_table  : array[timestamp];
ts            : timestamp;
my_index      : int;
my_part       : int;
in_progress   : LogOfUpdate_records;
lock          : LockType
```

Figure 4-2: State of an implemented service replica

4.1.2 Replica processing

The only type of processing that changes from the previous chapter is update processing. Gossip processing and garbage collection are exactly the same as described. Lookups are the same except that we decided to signal *unavailable* ("replica out-of-date") if the replica state is not recent enough instead of waiting. This way the client will know right away instead of possibly timing out waiting for the answer.

We implement the update operations as handler calls to take advantage of the transaction mechanism already in place. Recall that an action invoking a handler call creates a subaction for the call (the *call* action) and the actual invocation at the remote node runs as a subaction of the call action, so the invoking action will correctly inherit any locks the handler call obtains. In addition, implementing update operations as handler calls has the advantage of eliminating duplicate messages because we can abort the unsuccessful calls to replicas by aborting the call actions. This makes the remote action of an invocation an orphan (an action whose results are not wanted), so the orphan detection algorithm will take care of discarding any messages to it. This also works in the case of a crash and subsequent recovery, since the crash of the calling guardian's node will abort any outstanding handler call, making the corresponding remote action an orphan as well.

The handler operations of the replica guardian are called by the clients. A handler operation tries to get the replica lock by invoking the test_and_write operation on it. If it does not get the lock, the operation signals *unavailable* ("replica in use"), and the client must try somewhere else or again later. When an operation obtains the replica lock, it creates an update record and puts it into the *in_progress* log with the returned timestamp. We create the returned timestamp by incrementing *my_part* and storing it into a copy of the current timestamp.

For the *rebind* operation, we also made the decision to signal *unavailable* ("replica out-of-date") in the case that the replica's state is not recent enough instead of waiting. Since the *rebind* operation locks the replica, waiting could possibly tie up the client for a long period of time and could prevent another client from making use of the replica. The operation handlers are implemented as follows:

```
enter_guardians = proc (gset:SetOfGids) returns (timestamp)
        signals (unavailable (string))

    if test_and_write (S.lock) then
        S.my_part := S.my_part + 1;
        op_ts : timestamp := S.ts;
        op_ts[S.my_index] := S.my_part;
        add_entry (S.in_progress, ⟨op_ts, make_enter(gset)⟩);
        return (op_ts);
    else
        signal unavailable ("replica in use");
    end; % if

    end enter_guardian

delete_guardian = proc (g:guardian_id) returns (timestamp) signals (unavailable (string))

    if test_and_write (S.lock) then
        S.my_part := S.my_part + 1;
        op_ts : timestamp := S.ts;
        op_ts[S.my_index] := S.my_part;
        add_entry (S.in_progress, ⟨op_ts, make_delete (g)⟩);
        return (op_ts);
    else
        signal unavailable ("replica in use");
    end;

    end delete_guardian
```

```
rebind = proc (gm:MapOfGids,hm:MapOfHids,t:timestamp) returns (timestamp)
        signals (unavailable (string))

    if S.ts ~≥ t then signal unavailable ("replica not up-to-date") end;

    if test_and_write (S.lock) then
        S.my_part := S.my_part + 1;
        op_ts : timestamp := S.ts;
        op_ts[S.my_index] := S.my_part;
        add_entry (S.in_progress, ⟨op_ts, make_rebind (
            make_rebind_entry (gm,hm))⟩);
        return (op_ts);
    else
        signal unavailable ("replica in use");
    end; % if

    end rebind
```

After the update transaction has committed, the log entry is available to the rest of the replica guardian and is processed by a background process. The background process removes entries and does the operations in the same manner as gossip processing except that it does not have to check the timestamp of the entry and runs in a transaction. (This is so we do not lose the information if an entry is removed but not processed when a replica crashes.) Doing the actual work after a transaction commits does mean that there is a delay from the time an operation's transaction commits to the time the operation actually takes effect, but this delay should be small if we run the background process frequently. If a *lookup* or *rebind* operation should happen to arrive at the replica where the update was done but not processed and depends on it, the replica would answer as if it had not heard about the update yet.

4.2 Alternate solutions

In this section, we consider some alternate solutions to the serial solution. The difference between the solutions is in when and how the update information is processed. There are basically two kinds of solutions: those that invoke server operations before the commit of the calling transaction (*pre-commit* solutions), and those that invoke operations after the commit of the calling transaction (*post-commit* solutions). The *commit point* of the transaction is the point at which it can no longer be aborted.

Argus uses a standard two-phase commit protocol [6]. Each transaction has a set of guardians that are *participants* in the transaction. One is distinguished as the *coordinator*. The protocol has two phases: the *prepare* phase and *he *commit* phase. The protocol for a committing transaction is as follows:

- Prepare phase:
  1. The coordinator begins by sending *prepare* messages to all the participants.
  2. When a participant receives a *prepare* message, it writes out the new versions of changed objects to stable storage, and writes a *prepare* record to stable storage. Then it sends back a *prepared* message to the coordinator.
- Commit phase:
  1. When the coordinator has received a *prepared* message from all of its participants, it writes a *commit* record to stable storage. Then it send *commit* messages to the participants.
  2. When a participant receives a *commit* message, it installs any new versions as the current versions, and writes out a *committed* record to stable storage. Then it sends back a *committed* message to the coordinator.
  3. When the coordinator has received a *committed* message from all of its participants, it writes a *done* record to stable storage.

The commit point of a transaction is when the coordinator writes the *commit* record for that transaction to stable storage. After this point a transaction cannot be aborted.

For each of the solutions, we will first present the solution, sketch an implementation, and then evaluate the costs and problems of the solution. We will evaluate the costs of the different solutions using the following criteria:

- The amount of extra storage needed.
- The amount of extra computation.
- Ease of programming the solution. Related to this is the compatibility of the solution with the current run-time system. This includes the extent of changes to the existing run-time system.
- Other advantages and disadvantages.

4.2.1 Pre-commit solutions

Pre-commit solutions can be further divided into two types. An update operation can be called during the transaction or during the prepare phase of the commit protocol. The serial solution is a pre-commit solution of the first type. In this section, we present two other pre-commit solutions. The first one is an extension of the serial solution to allow concurrent updates. The second one is a solution of the second type. The update operations are called by the coordinator as part of the prepare phase of the commit protocol.

4.2.1.1 Local multipart time solution

As discussed earlier, concurrent updates cause a problem because smaller timestamps can be assigned to operations that commit after operations with larger timestamps. The serial solution deals with this by only running one update at a time. Another way of dealing with the problem is to generate incomparable timestamps at a single replica. Then it would not matter in which order the operations committed. A way of doing this is to make the logical time at a replica also be multipart. For example, the time at a replica could proceed: 1:1, 1:01,1:001, 1:0001, 2:1, 2:01, 3:1, 4:1, 4:01, etc. The meaning of these times is that the operations associated with times of 2:1 and 2:01 are strictly later than those operations with times beginning with 1, strictly earlier than the operations associated with times beginning with 3 or more, and concurrent with each other.

The clock at a replica produces time of the form $n:p_1..p_i$, where n is an integer and p_1 to p_{i-1} are 0 and p_i is 1. The clock is incremented in the following manner. Suppose the clock currently reads $n:p_1..p_i$. If there is an update operation already in progress then the next time is $n:p_1..p_ip_{i+1}$, where p_1 to p_i are 0 and p_{i+1} is 1. If there is no update operation in progress then the next time is (n+1):1. For example, if the clock read 3:001 when an update operation is invoked, then if another update operation is already in progress, the next time is 3:0001. If there is no other update operation in progress, the next time is 4:1.

We have to be able to compare and merge multipart times. We compare two multipart times as follows: $n:p_1..p_i$ is less than $m:r_1..r_j$ if $n < m$ or if $n = m$ and $p_k \leq r_k$, $k = 1$ to $\max(i,j)$ and at least one p_k is strictly less than r_k. (If $i \neq j$, then the shorter time is padded with zeroes.) If neither time is less than the other, then they are incomparable. We merge two multipart times, $n:p_1..p_i$ and $m:r_1..r_j$, as follows. If they are comparable, the merge is the greater of the two, for example, merge(1:1, 2:001) = 2:001. If they are incomparable, then the merge is $n:q_1..q_k$, where $q_k = \max(p_k,r_k)$, $k = 1$ to $\max(i,j)$. For example, merge(2:01,2:00001) = 2:01001.

This solution has extra computational overhead to determine the next timestamp and storage overhead for (physically) larger timestamps. Otherwise, it would be implemented in the same manner as the serial solution. We would get rid of the replica lock; the *in_progress* log already supports concurrent updates. The major programming effort would be in the implementation of the logical clock.

This solution suffers from potentially unbounded values if we get particularly bad executions where many operations overlap. To make this solution practical, we would have to assume that the system is loaded such that we will not get too many concurrent operations, or set a bound on the number of concurrent operations allowed. Note that a bound of one would make this solution exactly the same as the serial solution.

4.2.1.2 Prepare-phase solution

We can wait until the prepare-phase of the updating transaction to call the server operations. In this scheme, we have the coordinator of the transaction calling the appropriate operation at the same time as sending the *prepare* messages. The replica itself would be implemented as in the serial solution except that when an operation is invoked, the replica treats it as an operation invocation and *prepare* message combined. That is, the replica writes the new versions to stable storage directly. Then the called replica becomes a participant in the commit phase of the protocol. The coordinator sends back the returned timestamp to its participants as part of the *commit* message. The information the coordinator needs to be able to call the appropriate operation can be piggybacked with the system messages supporting transactions, such as handler reply messages. The coordinator would need storage to keep track of the operation that needed to be called, and then actually calling it. This would require reprogramming the commit protocol.

This solution holds the replica lock only during the commit protocol. This is an advantage if transactions are long, and many other transactions want to use the service. It also lessens the likelihood that a transaction will abort solely because of a server crash.

4.2.2 Post-commit solutions

Invoking update operations after a transaction has committed is a natural choice. The events that cause changes to the server have completed and will not be undone. This eliminates the need for atomic state and eliminates the concurrent update problem since the changes will not be undone. We can run the operations and assign the timestamps at the same time. Another advantage is that the server would not be called if the update transaction aborted.

The price for making the replica implementation easier is added time and complexity for the system. Since the commit point of a transaction is after the prepare phase is completely over, we have to wait to call the service operation at least until then, and then wait for the service operation to return before passing on the result of the operation to the participants of the transaction. We also have to check for duplicate messages since there would be no way to discard them automatically.

Post-commit solutions can also be divided into two types: those that call update operations during the commit phase of the commit protocol and those that call operations after the protocol has run. This section presents one of each type. The first has the operations called by the coordinator as part of the commit phase. The second has the operations called by the system after the replacement transaction has completed.

4.2.2.1 Commit-phase solution

A straightforward commit-phase solution would be to have the operations called by the coordinator after the *commit* record has been written but before sending out *commit* messages. The participants can tell the coordinator what needs to be done by piggybacking this information onto the *prepared* messages. The coordinator would send the returned timestamp to the participants with the *commit* messages.

Like the prepare-phase solution, the coordinator would need storage to keep track of what operation needs to be called. In addition, the coordinator would have to keep this information on stable storage in case it crashes after writing the *commit* record but before calling the server operations. This solution also would entail reprogramming the commit protocol.

The time cost of this solution comes in the form of a third phase waiting for the service operations to return, making the commit prctocol longer. This solution delays the commit of the transaction at the participants, because the coordinator cannot send the *commit* messages until after the update operation returns.

4.2.2.2 Non-transaction solution

We could simply wait until after the replacement or destroying transaction has completed to notify the server. This solution would operate in the same manner as the second caching strategy described in Section 3.4.1 at the guardian manager. The guardian managers at the affected nodes would have to keep information about the rebinding. If a guardian manager receives a call to a non-existent guardian, it would first look in its own state for the information and return it as part of the raised exception. If the server has not been notified, the guardian manager would raise a different exception than one that would be used after the server has been notified. This is because there would be no timestamp to return since the operation has not been done yet. Regular guardians would have to be changed to accommodate this new exception, but otherwise, would work the same way.

The difference between the caching strategy in Section 3.4.1 and this solution is that the information must be kept on stable storage until the server is notified. This is needed in case the replacement system is slow in notifying the server, so that calls to rebound guardians are properly handled if the guardian manager crashes and recovers before the replacement system notifies the guardian manager that the service has been told. After the server is notified, the information can be moved into the volatile cache or forgotten, whichever is the usual case.

4.3 Conclusions

4.3.1 Evaluation of serial solution

The serial solution means there is always one more participant in the commit protocol for a replacement transaction. The main advantage of the serial solution is that it is straightforward to program. It takes advantage of the existing transaction system, and integration into the current system is fairly simple.

The serial solution is affected by the "window of vulnerability" problem of the commit protocol. If the coordinator of a transaction crashes after writing out the *commit* record but before sending out *committed* messages, the replica will be unavailable for future update operations for the duration of the crash since it will not be able to find out the outcome of the transaction, and the unfinished transaction will still be holding the replica lock. However, lookups can still proceed and gossip can still be received and processed, so the situation is better than if the entire replica had crashed. A replica can also be tied up for long periods of time if a transaction is long or slow. This also increases the chance that a transaction may have to abort solely because of a replica crash. Concurrency for updates is limited to the number of replicas in the server, but this may be enough.

4.3.2 Comparison to alternative solutions

If we were building the Argus run-time system from the beginning, we might be inclined to choose the prepare-phase solution since the server really is part of the commit procedure. The prepare-phase solution does not slow down the protocol since the participants write new versions of stable data to stable storage at the prepare phase anyway. It would also lessen the probability that a transaction would abort solely because a server replica crashed. But given that Argus already exists, and the commit protocol is complex, the prepare-phase solution is not particularly attractive because it would require many changes to the current Argus commit protocol.

The multipart local time solution is conceptually the same as the serial solution. The logical clock is more complex than the simple counter of the serial solution. This size of the multipart times can become very large if many operations overlap. If we assume that operations do not overlap much, it is not clear that the multipart local time solution is much better than the serial solution given the added complexity.

The commit-phase solution seems to be rather worthless. Although it allows us to simplify the replica, it would really slow down the commit protocol, since it would wait for all of the *prepared* messages to arrive at the coordinator, and then for the service operations to return, before sending back the *commit* messages. The advantages from the commit-phase solution are also present in the non-transaction solution, whose disadvantages are not as severe.

In retrospect, the non-transaction solution might have been a better choice than the serial solution. The guardian manager shoulders more of the responsibility for keeping track of the information needed, but the guardian manager should cache some of the information anyway for efficiency reasons. This solution is better than the commit phase solution because it does not slow down the commit protocol. The non-transaction solution is also in keeping with the idea of lazy propagation of information found in most of the Argus system.

Chapter Five

Reconfiguration of the Server State

A configuration of the location server consists of the names and locations of the replicas that make up the server. The previous two chapters assumed that the server had a fixed configuration. The server was created once, and the replicas remained where they were created. In this chapter, we explore the possibility of reconfiguring the server. That is, we would like to be able to change the number or locations of the replicas that make up the server. We might want to move individual replicas for the same reasons as allowing objects to move, for example, because a node crashes too often or will be inaccessible for a long period of time. In addition, we would like to be able to *scale* the service to meet the requirements of availability and efficiency if the system changes in size. For example, if the system doubled in size, we might want to double the number of replicas in the server to keep the average number of queries and updates per replica roughly the same. On the other hand, if the system shrank by half, we might want to remove half of the replicas to avoid underutilizing resources.

Reconfiguration is done in a transaction. Either the change is completed, or the old configuration remains valid. We call a replica in the new configuration a *current* replica, and we call a replica that was in the old configuration, but not in the new one, an *obsolete* replica. The service provides three new operations to support reconfiguration: a *get_state* operation to obtain the state of a replica, a *create_with_state* operation that creates a new replica with the state given to it as an argument, and a *change_configuration* operation to install a new configuration at the called replica.

A simple scheme to do reconfiguration is as follows:

1. Start a transaction.
2. Read the states of all the replicas in the old configuration.
3. Construct the complete current state of the service.
4. Create any new replicas with the complete current state as an argument and construct the new configuration.
5. Invoke the *change_configuration* operation at the current replicas with the new configuration as an argument.
6. Destroy the obsolete replicas.
7. Commit the transaction.

Update processing must stop during the reconfiguration transaction in order to get a complete current state. This scheme is precise; each configuration has a definite beginning state and ending state. A current replica always starts out with all of the information from the previous configuration.

However, this scheme has some disadvantages. It stops the service for the duration of the reconfiguration, but since reconfiguration is expected to be rare, we might be able to tolerate such a disruption of service. A greater disadvantage is that it requires all of the server replicas to be up and able to communicate with the reconfiguration program. Reconfiguration would not be possible if some of the replicas were crashed or if the network partitioned with replicas on both sides. We would like to be able to reconfigure the service without having to access all of the replicas in the old configuration.

Once we allow reconfiguration involving less than all of the affected replicas, we must decide on how many replicas actually must be involved. All of the newly-created replicas are included since this is when they come into existence. A majority of the replicas in the old configuration can be required if synchronization of reconfiguration transactions is a problem. Requiring a majority of the old replicas would prevent concurrent reconfiguration transactions. On the other hand, if reconfiguration is rare, and usually done after careful consideration by a system administrator, we could just assume that the system administrator will insure that reconfiguration transactions are synchronized properly. We will assume that this is the case. Then it is possible to not involve any of the replicas in the old configuration, depending on how we gossip the news of the reconfiguration.

In this chapter, we describe a scheme that requires at least one replica from the old configuration to participate in the reconfiguration transaction. The first two sections of this chapter present our reconfiguration scheme. Section 5.1 describes the extensions needed to support our reconfiguration scheme, and Section 5.2 presents the abstract implementation of the basic scheme. Section 5.3 briefly sketches an optimization for the basic scheme. In Section 5.4, we address the question of how clients find the service after the reconfiguration is done. We outline and compare several schemes for finding the location service.

5.1 Extensions to support reconfiguration

Basically, the steps in the reconfiguration transaction are the same as were shown above. However, instead of reading the states of all the replicas in the old configuration, we only have to read one (although we might want to read as many as possible and merge the states we receive). We call this replica the *participant* replica. We create any new replicas with the state from the participant replica and construct the new configuration. Then we invoke the *change_configuration* operation at all newly-created replicas and the participant replica.

Replicas not participating in the reconfiguration transaction hear about the reconfiguration through gossip. The *change_configuration* operation is treated like any other update operation, and an update record with a change entry is added to the *send_gossip* list for it. When a non-participant replica encounters an update record for a configuration change, it reconfigures itself appropriately.

It is important to note that since we do not construct a complete current state, the state of the service stored at the current replicas may not have all of the information entered prior to the reconfiguration transaction. This means that an obsolete replica must continue to exist after it finds out about the reconfiguration and must send gossip to the current replicas in order to propagate any missing information that it holds.

Thus a reconfiguration is not complete until all of the replicas in the old configuration know about it and all of the information from the old configuration has propagated to the new configuration. While this propagation of information is taking place, the service is in a *hybrid* state between configurations. Our problem is determining how replicas must behave during this period.

In this section, we consider how to support this reconfiguration scheme. There are four issues to discuss:

1. How to identify replicas.
2. How to relate the states (and timestamps) from different configurations.
3. How to propagate information from the old configuration to the new configuration and how to determine that all of the information has propagated.
4. How to determine when an obsolete replica can be destroyed.

In the rest of this section, we state how we will identify replicas. Then we explain the use of version numbers to distinguish states from different configurations. Third, we address the problems in gossip processing in the hybrid state. Finally, we deal with returning to normal processing and destroying obsolete replicas.

5.1.1 Identifying replicas

With a fixed configuration, there is a one-to-one correspondence between a replica and its index into the configuration. We can ignore the actual names of the replica guardians, since it is sufficient to know just the index of a particular replica. In effect, a replica's index can be used as its id since they are unique. Now that we can change the configuration of the service, this is no longer the case. The index of a replica into a configuration and the replica at a particular index may change after a reconfiguration, so we need another way of uniquely identifying replicas. Since a replica is also a guardian, it has a unique guardian id assigned to it by the Argus system. We will use this id as the replica id. A configuration is an array of replica ids. The index of a replica into a configuration can be determined by matching the replica's id with the replica ids in the configuration. However, it is still convenient to retain the index of a replica as part of its state.

5.1.2 Version numbers

The second problem is how to relate states from different configurations. We would like a state from a later configuration to have a larger timestamp than a state from an earlier configuration. But a timestamp from a later configuration may have a different number of parts than a timestamp from an earlier configuration. And even if timestamps from different configurations have the same number of parts, their meanings are different since they refer to a different set of replicas. We must be able to distinguish between timestamps from different configurations and have some way of comparing and merging them.

We solve this problem by numbering the successive configurations of the service in increasing order. We call this number a *version_number* and prefix it to all the timestamps sent out by the replica. In other words, the timestamps in the server are now ⟨version_number, timestamp⟩ pairs. Timestamps with higher version numbers are later than all timestamps with lower version numbers. We redefine the timestamp operations on these new timestamps as follows:

```
new_merge = proc (ts1,ts2:new_timestamp) returns (new_timestamp)

    if ts1.version = ts2.version
        then return (⟨ts1.version,merge (ts1.timestamp,ts2.timestamp)⟩) end;
    if ts1.version > ts2.version
        then return (ts1)
        else return (ts2) end;

    end new_merge


new_equal = proc (ts1,ts2:new_timestamp) returns (bool)

    if ts1.version_number = ts2.version_number
        then return (ts1.timestamp = ts2.timestamp)
        else return (false) end;

    end new_equal


new_lt = proc (ts1,ts2:new_timestamp) returns (bool)

    if ts1.version_number = ts2.version_number
        then return (ts1.timestamp < ts2.timestamp) end;
    if ts1.version_number < ts2.version_number
        then return (true)
        else return (false) end;

    end new_lt
```

For the rest of this chapter, "timestamp" refers to this new type of timestamp. We will call timestamps with the same version number as the current configuration *current* timestamps. Timestamps with version numbers less than the version number of the current configuration are *old* timestamps.

5.1.3 Gossip processing in the hybrid state

In this section, we describe how gossip processing is done in the hybrid state. Recall that we make the simplifying assumption that all gossip from the old configuration has propagated to the new configuration before the next reconfiguration transaction is begun. This is a reasonable assumption since we expect reconfigurations to be done infrequently. In addition, we will also assume that the update entries of gossip from the old configuration are garbage collected from a replica's *send_gossip* list before the next reconfiguration transaction is begun.

To support reconfiguration, a replica can have one of five statuses. We represent a replica's current status by the state component, *status*. *Status* is a variant (a mutable discriminated union); its type is:

```
status_type = variant [normal       : null,
                       obsolete     : array[bool],
                       current      : current_status,
                       new          : null,
                       no_gossip    : null]

current_status = record [old_con       : configuration,
                         old_ts        : timestamp,
                         acknowledge   : array[bool],
                         received      : array[bool]]
```

If there is no reconfiguration going on, a replica has a status of *normal*. *Normal* status is the usual status of a server replica. It means that the replica is current and all information from any previous configuration has propagated to it.

When a replica hears about a reconfiguration (either through the *change_configuration* operation or through gossip) its status either changes to *obsolete* (if it an obsolete replica) or to *current* (if it is a current replica). The *obsolete* status means the replica has heard about the reconfiguration and is obsolete. A status of *current* means that the replica is current and there is a reconfiguration going on. When the reconfiguration finishes, replicas with *current* status change to *normal* status. Replicas with *obsolete* status are eventually destroyed. The meaning of the field components in the *obsolete* and *current* status cases will be explained later.

A status of *new* means that the replica is a newly-created replica. A replica with *new* status will always change to *current* status when the *change_configuration* operation is invoked at it. The meaning of the *no_gossip* status will be explained later.

To accomplish a reconfiguration, we must propagate the news about the reconfiguration to replicas that did not participate in the reconfiguration transaction. Current replicas will automatically find out through normal gossip since all replicas gossip to current replicas. Obsolete replicas that did not participate also need to be told. This is done by having replicas with *current* status gossip about the reconfiguration to obsolete replicas as well as current ones.

In addition, we must make sure that all of the information from the old configuration propagates to the new configuration. As stated before, obsolete replicas may have information that has not propagated to the current replicas, so they must gossip to the replicas in the current configuration. Also, current replicas that were part of the old configuration may have information from the old configuration that other replicas have not heard. The timestamps associated with such information will be old. We call this information from a previous configuration "late" gossip.

From a replica's point of view, a reconfiguration is finished when it has received all of the information from the previous configuration. We use the field components in the *current* status case to keep track of the information needed to determine when this happens. The entire reconfiguration process is completed when all current replicas have received all of the information from the previous configuration.

There are two problems for replicas with *current* status related to late gossip. The first is determining if the late gossip contains new information. We cannot use the normal gossip processing algorithm because the late entries will have old timestamps, while the replica's timestamp will be current. Since all old timestamps are less than current timestamps, gossip entries having old timestamps would be thrown away by the receiving replica even though they may contain information that the replica does not know about.

We solve this problem by having replicas save information about the old configuration when they change to *current* status and using this information to determine whether late gossip entries contain new information. Specifically, we save the old configuration in the *current* status field component *old_con* and the old timestamp in the *current* status field component *old_ts*. When a late gossip message is encountered, its timestamp is compared to *old_ts*. If it is less than or equal to *old_ts*, the message is discarded. If it is not less than or equal to *old_ts*, the entries are processed as usual using *old_ts* rather than *ts* to determine if the entry itself is new or old information. After the message has been processed its timestamp is merged into *old_ts*.

Late gossip entries can also arrive in gossip messages with current timestamps. (For example, the sending replica heard the information before it heard about the reconfiguration.) Since a replica does not know if there are any late entries in the message, it must look at all the entries in gossip messages with current timestamps, even if the message timestamp is less than or equal to the replica's timestamp. When a late gossip entry is encountered, its timestamp is compared to *old_ts*. If it is less than or equal to *old_ts*, the entry is discarded. If it is not less than or equal to *old_ts*, the entry is processed as usual and its timestamp is merged into *old_ts*. Gossip entries with current timestamps are processed normally.

The second problem is determining when a replica has received all of the information from the old configuration. We note that the change entry for the reconfiguration for a particular replica will follow all of the update entries of operations invoked before the reconfiguration in that replica's *send_gossip* list. If there is no garbage collection of the *send_gossip* list during the reconfiguration, then a replica knows it has received all of the information about operations invoked at a replica during the previous configuration when it encounters the change entry for that replica. The *send_gossip* list of the original sending replica reflects this order and processing by other replicas will retain this order.

We keep track of which replicas from the old configuration have gossiped a change entry in the *current* status field component *received*, a boolean array. The indices of *received* correspond to the *index*es of the replicas in the old configuration. Initially, the entries are all false. *Received[i]* is set to true when the replica encounters a change entry from replica i of the old configuration. When a replica receives a change entry for all of the replicas in the old configuration (*received[i]* is true for all i), it has heard all of the updates with old timestamps.

The solution only works if we do not garbage collect the *send_gossip* list of the replicas in the old configuration after the reconfiguration starts, since otherwise the replicas not participating in the reconfiguration transaction can garbage collect information from their *send_gossip* lists before it is propagated to newly-created replicas. In particular, it is important not to garbage collect the information that was not known by the participant replica when its state was read. The new replicas created with that state will never find out the new information if it gets garbage collected. We ensure that such information is not garbage collected in the following way:

1. The participant replica stops sending and receiving gossip and stops garbage collecting its *send_gossip* list when the *get_state* operation is executed.
2. Replicas with *obsolete* or *current* status stop garbage collecting their *send_gossip* lists.
3. When a replica with *normal* status receives a gossip message with a timestamp that has a version number greater than the replica's version number, it only processes a change entry (there must be one since this replica has not heard of the reconfiguration), leaving the gossip message on the message queue to be processed after the replica has reconfigured.

These rules effectively stop garbage collecting at replicas other than the participant replica when the reconfiguration transaction begins. The non-participant replicas cannot remove information before they find out that the participant replica knows about the information. The participant replica resumes sending and receiving gossip after it reconfigures (this is why it is included in the reconfiguration transaction). The participant replica's gossip message will have a timestamp with a higher version number than replicas that have not reconfigured. If a recipient of this message has not already reconfigured, this timestamp will cause it to look for the change entry first and reconfigure, which stops the garbage collecting. If the recipient has already *reconfigured*, then it has already stopped garbage collecting. We mark the participant replica by having the *get_state* operation change that replica's status to *no_gossip*.

5.1.4 Returning to normal processing and destroying obsolete replicas

A replica cannot garbage collect old information until all *current* replicas have heard its gossip. We can indicate the receipt of such information by having a current replica send an *acknowledged* gossip message to a replica when it encounters that replica's change entry. For a replica with *current* status, this means that it cannot return to *normal* status until it receives an acknowledgment from all of the other current replicas. For an obsolete replica, this means that it cannot be destroyed until it receives an acknowledgment from all of the current replicas.

We keep track of the acknowledgments in the status field component *acknowledge*, an array of boolean values. Both the *obsolete* status and the *current* status have this field component. The entries of the array are initially set to false. When an acknowledge entry from replica i of the current configuration is received, *acknowledge[i]* is set to true. When an obsolete replica has heard acknowledgments from all current replicas (*acknowledge[i]* is true for all i), it can be destroyed. A current replica can return to *normal* status when it has heard gossip from all the replicas in the old configuration and it has heard acknowledgments from all current replicas.

A replica must hear an acknowledgment from all current replicas directly before garbage collecting the old information because it cannot rely on a replica that has received its gossip to gossip the information to other replicas. This is because a replica may change to *normal* status upon the receipt of the gossip and garbage collect the late entries before sending the next round of gossip messages out.

5.2 Abstract implementation

In this section, we continue our abstract implementation of the service. We start from the abstract implementation of Chapter 3. We assume that all transactions commit and that the various processes in a replica run one at a time to completion. Recall that we also assume that all of the gossip from the old configuration has propagated to the new configuration before the next reconfiguration transaction is done. That is, all of the replicas in the current configuration have *normal* status when a reconfiguration is started.

We begin our implementation with a description of the new data structures needed to support reconfiguration. Then we give the code for the *change_configuration* operation and gossip processing. Finally, we discuss modifications to the service operations and garbage collection algorithms.

5.2.1 Data structures

Recall that timestamp refers to the new type of timestamps. Likewise, merge, equal, and lt refer to the operations new_merge, new_equal, and new_lt, respectively. In addition to the *status* data structure introduced in Section 5.1.3, we add the following data structures to the state:

My_id is the id of the replica. The id of a replica is assigned by the run-time system when a replica is created. It is available through a system primitive.

Current_configuration is the configuration currently in use. A configuration is an array of replica ids that make up the server. They are ordered such that configuration[i] is the id of the i^{th} replica of that configuration.

The update_records have two additional arms:

```
update_record = oneof [enter   : SetOfGids,
                       rebind  : rebind_entry,
                       delete  : guardian_id,
                       change  : change_entry,
                       ack     : null]

rebind_entry = struct[gm:MapOfGids, hm:MapOfHandlers]
change_entry = struct[new_con:configuration, new_version:int, id:replica_id]
```

The change arm represents a *change_configuration* operation. Besides the arguments to the *change_configuration* operation, the change entry also includes S.my_id of the replica that created the entry. The ack arm represents an acknowledgment from the sending replica.

Figure 5-1 summarizes the new replica state. *My_index* is no longer a constant. It is the replica's index into the current configuration and may change when a reconfiguration is done.

5.2.2 Changing configurations

The reconfiguration transaction can be viewed as having two parts. The first part is preparatory: we read a state, create the new replicas, and construct the new configuration. In the second part, we do the actual reconfiguration by invoking the *change_configuration* operation at the new replicas and the participant replica.

```
gmap                    : MapOfGids;
hmap                    : MapOfHandlers;
exists                  : SetOfGids;
deleted                 : SetOfDeletedGids;
send_gossip             : SetOfUpdate_records;
gossip_table            : array[new_timestamp];
ts                      : new_timestamp;
my_index                : int;
my_part                 : int;
status                  : status_type
my_id                   : replica_id;
current_configuration   : configuration;
```

Figure 5-1: State of a reconfigurable replica

Obtaining a replica state is done by invoking the *get_state* operation. The *get_state* operation also causes the replica to stop sending gossip and garbage collecting by changing the replica's status to *no_gossip*. It is implemented by:

```
get_state = proc ( ) returns (MapOfGids,MapOfHandlers,SetOfGids,SetOfDeletedGids,
            timestamp,configuration)

    change_no_gossip(S.status,nil); % change status to no_gossip
    return (S.gmap,S.hmap,S.exists,S.deleted,S.ts,S.current_configuration);

    end get_state
```

New replicas are created with the *create_with_state* operation. This operation takes the items returned by the *get_state* operation as arguments, creates a new server replica, and returns the newly-created replica's id. The new replica is created with a status of *new* and the arguments are used to initialize the corresponding parts of the new replica's state. *My_id* is set using a system primitive provided by the Argus run-time system. The rest of the replica state is left undefined. We can do this because a replica with the status of *new* will never be called by a client until after it has participated in the second part of the reconfiguration transaction, which will initialize the rest of the replica state. The operation is implemented by the following code:

```
create_with_state = proc (gmap:MapOfGids,hmap:MapOfHandlers,exists:SetOfGids,
          deleted:SetOfDeletedGids,ts:timestamp,con:configuration)
      returns (replica_id)

   S.status := make_new (nil);
   S.gmap := gmap;
   S.hmap := hmap;
   S.exists := exists;
   S.deleted := deleted;
   S.ts := ts;
   S.current_configuration := con;
   % The id of a guardian is set by the run-time system when the guardian is
   % created; a system primitive returns the id of the invoker.
   S.my_id := % system primitive
   return (S.my_id);

   end create_with_state
```

Change_configuration changes the configuration of a replica and saves the appropriate information depending on whether the replica is current or obsolete. We assume that the new configuration has a version number greater than the current one. (In a real implementation, we would check this.) We implement the operation in the following manner:

```
change_configuration = proc (new_configuration:configuration, new_version:int)

   num_old : int := number of replicas in old configuration;
   num_new : int := number of replicas in new configuration;

   acknowledge : array[bool];
   for i := 1 to num_new do acknowledge[i] := false end;
   if S.my_id $\in$ new_configuration
      then % a current replica
         received : array[bool];
         for i := 1 to num_old do received[i] := false end;
         if S.my_id $\in$ S.current_configuration
            % part of the old configuration, have heard everything from self
            then received[S.my_index] := true end;
         S.my_index := index of S.my_id in new_configuration;
         acknowledge[S.my_index] := true % have heard from self
         change_current(S.status,make_current_status(S.current_configuration,
            S.ts,received,acknowledge));
         S.ts := $\langle$new_version,$\langle 0_1,\ldots,0_{num\_new}\rangle\rangle$;
         S.ts[S.my_index] := 1;
         S.my_part := 1;
         for i := 1 to num_new do
            S.gossip_table[i] := $\langle$new_version,$\langle 0_1,\ldots,0_{num\_new}\rangle\rangle$ end;
```

```
        else % an obsolete replica
            change_obsolete(S.status,acknowledge); % change status to obsolete
            S.my_part := S.my_part + 1;
            S.ts[S.my_index] := S.my_part;
        end; % if
    S.current_configuration := new_configuration;
    S.send_gossip := S.send_gossip ∪ {⟨S.ts,make_change(
        make_change_entry(new_configuration,new_version,S.my_id))⟩};

    end change_configuration
```

5.2.3 Gossip processing

As discussed earlier, all replicas send gossip to replicas in the current configuration. Replicas with *current* status also gossip to replicas in the old configuration that are obsolete.

Recall that a gossip message M has components *gossip_list*, *ts*, and *index*. We now add a component *id*, the id of the sending replica. We implement the gossip processing with the following code:

```
new_gossip_processing = proc ( )

    tagcase S.status of
        obsolete (acknowledge:array[bool]) : % obsolete replica
            for u ∈ M.gossip_list do
                tagcase u.rec of
                    ack :
                        % This will be a single entry message.
                        % Acks can only come from current replicas by assumption.
                        acknowledge[M.index] := true;
                    others:
                        % Take M off the message queue; replica can ignore anything else;
                        % this would a late gossip message.
                        return;
                    end; % tagcase
                end; % for
        current (c:current_status) : % current replica during reconfiguration
            if M.ts ≤ c.old_ts then
                % Take M off the message queue.
                % It is possible for this message to have a change entry for an
                % obsolete replica whose ack was lost; if this is so, the ack
                % will be sent in the normal case when this message is sent again.
                return;
                end;
```

```
         for u ∈ M.gossip_list do
             tagcase u.rec of
                 change (chg:change_entry) :
                     if (u.ts.version_number < S.ts.version_number ∧ u.ts ~≤ c.old_ts) ∨
                             (u.ts.version_number = S.ts.version_number ∧ u.ts ~≤ S.ts) then
                         % haven't heard this one yet
                         S.send_gossip := S.send_gossip ∪ {u};
                         if chg.id ∈ c.old_con then
                             % not from a newly-created replica
                             index : int := index of chg.id in c.old_con;
                             c.received[index] := true
                             end; % if
                         if u.ts.version_number < S.ts.version_number
                             % old timestamp, merge with old_ts
                             then c.old_ts := merge (c.old_ts, u.ts) end;
                         end % if
                     % always ack change entry
                     send {⟨S.ts, make_ack(nil)⟩} to chg.id;
                 ack :
                     c.acknowledge[M.index] := true;
                     return; % do not update gossip_table or replica timestamp
                 others :
                     if (u.ts.version_number < S.ts.version_number ∧ u.ts ~≤ c.old_ts) ∨
                             (u.ts.version_number = S.ts.version_number ∧ u.ts ~≤ S.ts) then
                         % haven't heard this one yet
                         S.send_gossip := S.send_gossip ∪ {u};
                         % Process entries the same as regular gossip processing.
                         if u.ts.version_number < S.ts.version_number
                             % old timestamp, merge with old_ts
                             then c.old_ts := merge (c.old_ts, u.ts) end;
                 end; % tagcase
             end; % for
         if (M.id ∈ S.current_configuration) ∧
                 (M.ts.version_number = S.ts.version_number) then
             % update gossip_table and replica timestamp if from a
             % current replica that knows about the reconfiguration.
             S.gossip_table[M.index] := merge(S.gossip_table[M.index],M.ts);
             S.ts := merge (S.ts, M.ts);
             end; % if
     normal : % current replica and no reconfiguration
         if M.ts.version_number > S.ts.version_number then
             % A reconfiguration has happened. Look for a change entry.
             for u ∈ M.gossip_list do
                 tagcase u.rec of
                     change (chg : change_entry) :
                         if chg.version > S.ts.version_number then
                             change_configuration (chg.new_con,chg.new_version)
                             % Leave M on the message queue.
                             return;
                             end;
                     others: % ignore the other entries
                     end; % tagcase
                 end; % for
```

```
        elseif M.ts.version < S.ts.version then
            % A delayed gossip message. Receiving replica will have already
            % heard the updates in this message, but the sending replica
            % may not have heard an ack yet, so send one.
            send {⟨S.ts, make_ack(nil)⟩} to M.id;
        else
            % Nothing unusual, process normally. Ack the change entries
            % (in case the previous acks got lost) and ignore the acks.
        end; % if
    no_gossip : % ignore gossip
    end; % tagcase

end new_gossip_processing
```

5.2.4 Other replica processing

In operation processing, there is a possible problem with an operation getting a later timestamp than another operation that happened after it in real time. For example, suppose an *enter_guardians* operation happens at replica in the new configuration and gets a current timestamp, and then the *delete_guardian* operation for one of the guardian ids happens at a replica that has not heard about the reconfiguration yet and gets an old timestamp. The enter timestamp of that guardian id will be greater than the timestamp of its deletion. This might cause the garbage collection algorithm to remove the entry from its *deleted* set too soon, creating an inconsistent state.

We solve this problem by having all service operations take an extra argument, the version number of configuration known to the client. There are three cases to consider: the client's version number is less than the replica's version number, the client's version number is equal to the replica's version number, and the client's version number is greater than the replica's version number. None of these cases apply to replicas with *new* status, since they will be changed to *current* status before any clients find out about them.

If the version number argument is less than the version number of the replica, the caller has not heard about the reconfiguration, and the replica refuses to do the operation. The operation signals an exception and returns the new configuration and version number. This is done by all replicas regardless of status.

A client's version number may be greater than the replica's version number if the client found out about the reconfiguration before the replica. This can cause the same problem for enters and deletes for a client that knows about the reconfiguration and does an enter getting a current timestamp, and then does a delete at a replica that does not know yet. The second replica would return an old timestamp. We handle this by having a replica signal *unavailable* ("replica out-of-date"), if its version number is not large enough. Note that this situation is only possible in the case of a replica with *normal* status. Clients that know about the reconfiguration will not call obsolete replicas, and such a call to a replica with *current* status would contradict our assumption of non-overlapping reconfigurations.

If the version numbers are the same, then both the client and replica know about the reconfiguration, and the operation can proceed as described before.

These tests would be implemented for all service operations in the following manner:

```
new_operation = proc (...,v:version_number)
        returns (...,timestamp)
        signals (...,new_configuration (configuration,int))

   tagcase S.status of
      obsolete :
         signal new_configuration(S.current_configuration,S.ts.version_number);
      current:
         if v < S.ts.version_number then   % client has old configuration
            signal new_configuration(S.current_configuration,S.ts.version_number) end;
      normal :
         if v < S.ts.version_number then   % client has old configuration
             signal new_configuration(S.current_configuration,S.ts.version_number) end;
         if v > S.ts.version_number then   % replica has old configuration
            signal unavailable ("replica not up-to-date") end;
      no_gossip :
         if v < S.ts.version_number then   % client has old configuration
            signal new_configuration(S.current_configuration,S.ts.version_number) end;
      end; % tagcase

   % Do the operation.  Returns S.ts.

   end new_operation
```

However, this is not enough for the *rebind* and *lookup* operations at replicas with *current* status. These operations rely on information in states with timestamps less than the argument timestamp to be present. This implies that a call with a current timestamp argument may rely on information from the old configuration. Since we do not know what the information is or which state it comes from, we must refuse the operation and signal *unavailable* ("replica out-of-date"). A replica can do rebinds and answer queries if the timestamp argument is old and the replica's old timestamp is large enough. (The timestamp argument of *rebind* and *lookup* operations may be an old one even if the client has heard about the reconfiguration and has the current configuration and version number.)

The *rebind* and *lookup* operations need the following additional tests in the *current* status case:

```
    :
    :
if t.version_number = S.ts.version_number  % not all of the old gossip in yet
    then signal unavailable ("replica not up-to-date") end;
if old_ts ~≥ t  % definitely do not have the old gossip needed
    then signal unavailable ("replica not up-to-date") end;
    :
    :
```

As stated before, garbage collection of the *send_gossip* list is not done in a replica with *no_gossip*, *current* or *obsolete* status to make sure that all of the information from the old configuration propagates to the new configuration. Garbage collection of the *deleted* set cannot be done in replicas with *current* status for similar reasons. The current timestamps in the gossip table may cause entries with old timestamps to be removed from the *deleted* set too soon, causing an inconsistent state. Both of these procedures would test the status of the replica before proceeding. Garbage collection of the maps is unaffected by the reconfiguration-scheme.

5.2.5 Returning to normal processing and destroying obsolete replicas

When a reconfiguration is finished, two things must happen. First, the current replicas must return to *normal* status. Second, obsolete replicas must be destroyed.

Reconfiguration_completed is invoked periodically by replicas with *current* status to check whether the replica can return to *normal* status. It sets S.status to *normal* when all the elements of *received* and *acknowledge* are true. It is implemented by the following procedure:

```
reconfiguration_completed = proc ( )

    tagcase S.status of
        current (c:current_status) :
            if ∀ i, c.received[i] = true ∧ ∀ j, c.acknowledge[j] = true
                then change_normal (S.status, nil) end; % change status to normal
        others : return;
        end; % tagcase

    end reconfiguration_completed
```

Destroy_obsolete is invoked periodically by replicas with *obsolete* status to check whether the replica can be destroyed. When an acknowledgment gossip message has been received from all current replicas, the replica is destroyed by invoking the **terminate** statement. It is implemented by the following procedure:

```
destroy_obsolete = proc ( )

    tagcase S.status of
        obsolete (acknowledge:array[bool]) :
            if ∀ i, acknowledge[i] = true then terminate end;
        others : return;
        end; % tagcase

    end destroy_obsolete
```

5.3 An optimization

In this section, we briefly sketch an extension to our basic reconfiguration scheme. It is an optimization for the case where the replicas of the new configuration are a subset of the old configuration. We call this a "benevolent" reconfiguration. Besides being a more efficient reconfiguration procedure, this extension also gives us a way of dealing with frequent or long-term node or network faults affecting service replicas.

Suppose that a service replica resides at a node that continually crashes or gets partitioned from the rest of the service replicas often. We would like to replace this replica with one on a more reliable node. We can run our basic reconfiguration scheme and eventually it will be done, but communicating with the obsolete replica may be intermittent causing it to continue to accept update operations (if the change entry does not arrive) or preventing the current replicas from changing to normal status (if the obsolete replica's gossip does not arrive). This may continue for a long period of time. Since most lookups have to wait for normal status, this could keep the service from answering lookups, effectively stopping the system.

A way to avoid this situation is to do a "benevolent" reconfiguration first. The basic idea in a benevolent reconfiguration is that if the new configuration is a subset of the old configuration, we can continue using the same version number and the same number of timestamp parts. This is because, in effect, we are keeping the same configuration, but some of the replicas are permanently unavailable. Operations at the current replicas can continue normally, so there is no disruption of service. Eventually, the obsolete replicas find out about the reconfiguration and stop accepting updates. When all of the current replicas receive a change entry for the reconfiguration from all of the obsolete replicas, the reconfiguration is done.

Returning to the scenario posed, we would first do a benevolent reconfiguration to take the troublesome replica out of the configuration. Then we would do a regular reconfiguration to add the replacement replica. The first reconfiguration does not disrupt service, and the second reconfiguration should be "easier" since we got rid of the potentially troublesome replica.

5.4 How clients find reconfigured services

To keep track of the reconfigurable service, a guardian manager keeps a stable copy of the current configuration and version number in addition to its stable timestamp. As discussed earlier, the service operations take the version number as an argument. The following four situations are possible when a service operation is invoked:

1. The guardian manager's version number is equal to the replica's version number. The operation returns as described previously. If it returns an unavailable signal, the guardian manager tries again later or at a different replica.
2. The guardian manager's version number is less than the replica's version number. The replica signals that there is a new configuration. The guardian manager writes the new configuration and version number to stable storage and tries the call again (at a replica in the new configuration).
3. The guardian manager's version number is greater than the replica's version number. The replica signals that its not up-to-date. The guardian tries again later or at a different replica.
4. A signal comes back indicating that the replica is non-existent. The guardian manager knows that the configuration has changed. It can try to find out what the new configuration is by trying the other replicas in its configuration. However, it is possible that all of the replicas in the configuration the guardian manager knows about have been destroyed.

Although it is likely that guardian managers are active enough to find out about reconfigurations as described, especially if there are common replicas in successive configurations, it is possible that a guardian manager will not find out about a new configuration before all of the replicas in the configuration it knows about are destroyed. This might happen if the node of a client is down or partitioned from the rest of the system for a long period of time and misses several reconfigurations. When such a situation happens, we need to have some way for clients to find the service. This section addresses this issue.

We cannot use the replacement method supported by this thesis to reconfigure the location service. (Clearly, we cannot use the location service to find the location service.) Other methods for allowing clients to find services that have moved fall into four general categories:

1. Notify a name service provided by the system.

2. Require a subset of replicas to be in every configuration.
3. Have clients broadcast a request for the current configuration.
4. Chain configuration information such that a client can follow the chain to the current configuration.

Each method has its advantages and disadvantages. One thing we would like to consider is whether the method will perform well in the presence of failures. Making the service highly-available and reconfigurable would not matter much if clients could not find it.

Notifying a name service just pushes reconfigurability to another service. We have the same problem if we want to reconfigure the name service. This solution also means that the ability of clients to find the location service depends on the availability of the name service. The system could have difficulty notifying the name service of the change in location service configuration, or the clients could have difficulty finding an available copy of the name service.

Requiring a subset of replicas to be in every configuration is feasible. Typically, we expect that successive configurations will have many replicas in common as we add or remove replicas from the service. But if the system runs for a long time, we can imagine that eventually all of the replicas we started with will be removed.

A real broadcast would be difficult and expensive in system with no broadcast primitives (like Argus). We can implement a "ask your neighbor" multicast like that proposed by Henderson [7]. A logical network system defining the concept of neighbors would have to be implemented on top of the Argus system. Such a scheme is still expensive, but if the protocol is not run often, the cost is amortized over the life of the system.

Chaining configuration information was proposed by Herlihy for general quorum consensus [8]. The basic idea is that obsolete configurations have pointers to the next configuration. There is a path from any obsolete configuration to the current configuration, and the client can follow the path. The obsolete configurations stay in existence until all clients know about the next configuration on the chain. Chaining seems to be a natural extension of our reconfiguration method. Adapted for the multipart timestamp technique, chaining would only mean keeping obsolete replicas around until all clients know about the next configuration. An obsolete replica already contains a pointer to the new configuration, and a client already gets the next configuration if it invokes an operation at an obsolete replica. The major problem with this scheme is that obsolete replicas take up resources and might be around for a long time. In addition, it is not easy to determine when all possible clients have heard about the next reconfiguration. Herlihy proposes a reference counting scheme to garbage collect the old configurations that we can also adapt to allow us to determine when an obsolete replica can be destroyed, but it is fairly complex.

Either broadcasting or chaining would be suitable for our purposes. Using a name service solves our immediate problem, but not the problem of finding reconfigurable services in general. The fixed subset solution places a constraint on system development: certain nodes must always exist. It may not be possible to meet such a constraint in the long run. Nodes get old and can become non-functional despite our best efforts.

If reconfiguration is rare and successive configurations usually have many replicas in common, it would be very unlikely that a client would miss enough reconfigurations to not know a replica that knows the current configuration. In this case, the amortized cost of broadcasting would be sufficiently low to be practical. In the same situation, chaining is expensive since we have to keep all obsolete replicas around even if only one client has not heard about the new configuration, and even if some of the replicas it knows about do know about the current configuration.

On the other hand, if reconfiguration is frequent and successive configurations have few replicas in common, it is more likely that a client could miss enough reconfigurations to not know a replica that knows the current configuration. Then the cost of broadcasting may become unreasonable as more clients must broadcast to find the service. In this case, chaining may be the better solution.

Chapter Six

Conclusions

This thesis described the design and construction of a location service to aid in finding objects that move in a distributed system. Recall that object movement can be viewed in two ways. In the "proper name" view, objects are denoted by unique identifiers that they keep as they move from node to node. The other view is that moving an object is replacing it with another object. That is, when we move an object we create a new replacement object, transfer the state of the old object to the replacement object, and then destroy the old object. The name of the old object becomes an alias for the new object. We chose the "replacement view" of object movement for our service because it is more general than the proper name view. The replacement view allows us to "split" or "merge" objects without revealing the change to the entities that access those objects. The location service records the aliasing of the old name to the new object. We call this "binding" the old name.

Our goals for the service were to make it highly-available and efficient. In particular, if the node at which an object resides is accessible then an entity wanting to access that object should be able to find it with high probability. We also had an implementation goal to insure that clients make progress as they make successive lookup requests of a particular handler name; if a client already knows that a certain address for a handler name is out-of-date, it should not receive that address as an answer to a lookup request of that handler name. To meet our goals, we needed to replicate the service state. Although there are many well-established replication techniques that would satisfy our needs, we chose to implement the service using a new replication technique, Liskov's multipart timestamp technique. This technique uses multipart timestamps and gossip messages to maintain a consistent state. One of the reasons for choosing this technique was to show that it was practical to use. An evaluation of the technique will be presented later.

Each replica in the service was implemented as an Argus guardian. The thesis presented an abstract implementation of a replica, describing the data structures and processing algorithms of the service. The problem with concurrent updates transactions was solved by making the replica state atomic and running the update operations at a replica serially. The update operations were implemented as handlers of the replica guardian to take advantage of the existing transaction system. Several other solutions to the problem were also presented and compared to the implemented solution. Some of these solutions might have been chosen under different circumstances.

In a long running system, the configuration of the system may change; therefore we might want to change the configuration of the location server as well by changing the number or location of its replicas. The thesis investigated extensions to the multipart timestamp technique to allow us to do reconfiguration.

In summary, the contributions of this thesis were:

- a location service for Argus and a basis for general object finding
- a practical application of the multipart timestamp technique
- extension of the technique to allow reconfiguration of the service state

The rest of this chapter evaluates the multipart timestamp technique as a method for constructing highly-available services, discusses some work related to the thesis in the area of locating objects, and suggests some areas of future work.

6.1 Evaluation of the multipart timestamp technique

One of the reasons for choosing the multipart timestamp technique as the replication technique for the location service was to show that a practical application could be built using the technique. Several examples of services using the technique have been proposed (for example, orphan detection [10], garbage collection in a distributed heap [18], and deleting old versions in a hybrid concurrency control scheme [27]), but none have been implemented.

The information kept by these servers can be characterized by the following properties:

1. Updates are idempotent. This means that it does not matter how many times an operation is executed because the effect is the same as if it was only executed once.
2. Updates do not need to be totally ordered. The technique has no way of ordering parallel updates that happen at different replicas. To do so would result in the loss of the advantages this technique has over traditional replication techniques.
3. Queries identify the updates whose effects should be reflected in the query result. This gives a replica a local way of determining whether it has the needed information.
4. The information states can be merged in a well-defined manner.

The location service operations for the most part fit these properties. The only problem was with *enter_guardians* and *rebind* operations involving the same guardian or handler. The lookup algorithm requires that the state it reads from contain the enters for the target guardian ids of any rebinds, since otherwise it would give the wrong information. (Recall that a guardian id is a target if it appears in the right component of a binding.) We satisfied this requirement by sending a timestamp known to be at least as late as the merge of the *enter_guardians* operation timestamps of the target guardian ids as an argument to the *rebind* operation and requiring that the returned timestamp to be at least as late. This causes the replica at which the *rebind* operation occurs to wait for the information about the *enter_guardians* operations before processing the *rebind* operation. A lookup at that particular replica will have the correct state to answer a request about that binding. This replica's gossip will reflect that state as well so all other replicas will know about the enters if they know about the rebind.

In general, we would not want an update operation to have to wait for another update operation to take effect before it could be done. In the case of the *rebind* operation, however, we expect the *enter_guardians* operations to have been done far enough in the past to have propagated to all replicas in a normally running system. Thus the *rebind* operation should have to wait only when there are failures. An alternate strategy would be to do the rebind operation without waiting and delay the lookups until the condition is met.

We can compare the performance of the multipart timestamp technique with other types of replication techniques. For example, a large number of replication techniques can be classified as voting techniques. Included in this class of techniques are the original weighted voting scheme by Gifford [5] and its generalizations such as general quorum consensus [8]. In the simple case of voting, the set of replicas visited by operations that read the service state must intersect with the set of replicas that are visited by operations that modify the service state. Voting is available in the presence of failures as long as a client can access the correct number of replicas.

It should be noted that voting techniques can support consistency constraints that the multipart timestamp technique cannot. For example, voting techniques can be used for applications that require read operations to return the most recently written value. For the location service, it is not always necessary to have the most recent state to answer a particular lookup request since the most recent update to the state may not affect the answer. In addition, we only require that a client be able to make progress when it makes successive lookup requests of a particular handler name. The multipart timestamp technique satisfies this weaker requirement.

We prefer the multipart timestamp technique over voting techniques for several reasons. It is more efficient and available than voting techniques when the system is running normally. Since with voting techniques, the set of replicas accessed by update operations must intersect with the set of replicas accessed by query operations, the availability and efficiency of update operations are inversely proportional to the availability and efficiency of query operations. Voting techniques can make one kind of operation as available and efficient as the multipart timestamp technique, but not both kinds of operations. The multipart timestamp technique allows both lookups and updates to happen at just one replica.

The multipart timestamp technique provides more availability than voting techniques in the presence of partitions. *Enter_guardians* and *delete_guardian* operations can proceed as long as one replica is accessible to the client. (Note that nothing less than one replica per node can prevent the situation of a client being isolated from all replicas.) Lookup requests may be delayed if the lookup requires information about an update that was processed on the other side of the partition. This might happen if the replica that processed the update was separated from the client and the other replicas after sending a response to the client, but before it communicated with other replicas. This situation is unlikely if gossip is frequent. If we need to make it even less likely, then we can gossip before the transaction commits. This causes some problems (such as, how to undo the operation if the transaction aborts) and lessens the availability of update operations (more replicas must be accessible in order to do the operations), so a choice must be made in trading off complexity and availability of updates versus the acceptable probability of being unable to process some lookup requests for some period. However, even if we do require more than one replica to know about an update before responding to the client, our update operations will be more available than those using voting techniques because we will not need as many replicas to participate in the operation. A *rebind* operation can also be delayed, but as argued before, it is likely that the *enter_guardians* operations that the *rebind* operation relies on will have propagated to all replicas.

The multipart timestamp technique provides more availability than voting techniques in the presence of crashes. In voting techniques, operations become unavailable if too many replicas crash. In our service, the only time operations become unavailable is if all of the replicas in the server crash. We only have a problem if a particular replica crashes before it gossips an update that it has processed. We can make that possibility arbitrarily small by means of the same trade-offs as discussed for partitions.

The multipart timestamp technique also scales up better than voting techniques. If we add more replicas to a voting technique, then at least one kind of operation will be less available, since it will have to access more replicas to complete it. The multipart timestamp technique maintains the same availability for each operation, regardless of the number of replicas.

We can also compare the multipart timestamp technique with techniques that guarantee weaker consistency constraints. The usual reason for having weaker constraints is to increase availability in the presence of failures. An example of a system using such a technique is LOCUS, the network operating system developed at UCLA [24]. LOCUS gives direct support for replicated files to increase availability. While the system is functioning normally, LOCUS maintains a consistent state among the copies of the file. However, during a network partition, it allows inconsistencies in file copies to develop. These inconsistencies are detected and reported when the network rejoins. Any resolution is done at the application level.

To detect inconsistencies in file versions after a partition is repaired, LOCUS uses version vectors. A version vector is a mapping of a node to the number of times a file has been updated at that node. Incomparable version vectors indicate version conflicts in the various copies of the file. The version vectors are similar to the multipart timestamps that are used in the multipart timestamp technique. Multipart timestamps essentially also map a replica to the number of updates done to the service state at that replica. However, we use the multipart timestamp to prevent data inconsistencies, not just to detect them after the fact. We can do this because we know the meaning of the data in the service state and can merge the information. By contrast, arbitrary files do not have such a property. The multipart timestamp technique gives us nearly the same amount of availability as in LOCUS (there will be few situations when an operation cannot be done), without the complexity of fixing inconsistencies.

6.2 Related work

There have been several proposed methods of finding objects that move. In this section, we discuss three of these methods: forwarding addresses, searching, and establishing a rendezvous. We will call an entity that is trying to access an object a *user* of that object.

6.2.1 Forwarding addresses

Fowler [4] proposes a method based on proper names and forwarding addresses. His model of computation is the same as ours, but his objects retain their identities when moved. In his scheme, an object is known by a proper name that always denotes that object. Basically, every time an object moves, it leaves a forwarding address at its former residence. When a user wants to access an object that has moved, this chain of forwarding addresses is followed from the last known address until the object is reached or it can be ascertained that the object has been destroyed. To make this scheme practical, three path compression algorithms are given to be applied to the chain of forwarding addresses. This solution is completely decentralized; no one entity knows where all of the objects are.

Fowler's solution does not tolerate failures as well as our method. If there is a failure along an object's forwarding address chain, a user may not be able to find it. His solution to this problem is to relate directly the availability of his algorithm for finding a particular object to the availability of the object itself. The algorithm will guarantee to find an object in the presence of k failures only if the object itself can tolerate k failures. That is, it must be the case that k failures will not make all copies of an object inaccessible. This implies that there must be at least k+1 copies of the object in the system and that k failures cannot partition the network. The scheme is to allow each copy to move independently, so effectively there are k+1 separate forwarding address chains that can be followed. Since k failures cannot partition the network or break all of the forwarding address chains, the algorithm will find the object. Note that this scheme means that Fowler's algorithm is not fault-tolerant for non-replicated objects. Such an object would only have one forwarding address chain and any failure along that chain could lead to the situation where the non-replicated object is on a node accessible to the user, but the user cannot find the object.

We believe that the availability of finding an object should not depend on an object's implementation. The applications programmer should not have to replicate an object just to accommodate being able to find the object after it moves. Our method does not rely on an object being replicated in order to provide availability of the service. The only time our method must delay a lookup request for a long period of time is if the information needed is currently inaccessible due to it being on a crashed node or on the other side of a partition from the user. As discussed before, we can make the probability of this happening arbitrarily small.

Another advantage that our method has is that garbage collecting the information relating to destroyed objects is easier. In both schemes, information about an object must be kept until the object is destroyed. However, in Fowler's scheme, the information may be present at many nodes (a long-lived, frequently moved object would be such a case). Because the information about an object is spread out in the system, garbage collection in the forwarding address scheme is slow. A node must wait until an access returns an exception for a non-existent object or it is told by another node that tried to access the object that it has been destroyed. If no user tries to access the object, a node may keep the information forever. (Actually, a node needs to keep forwarding address information only until all users of the object know about the new address, but knowing when this condition becomes true is a difficult task.) Garbage collection in Fowler's scheme could be done similarly to ours if the forwarding address entries also kept backward pointers to the previous forwarding address entry. Then when an object is destroyed the backward chain can be followed to notify the nodes where the object used to reside that it has been destroyed. Garbage collecting in our scheme is easier and more straightforward because of the centralization of the information.

6.2.2 Searching

Henderson [7] proposes finding objects by searching for them. Her model of computation is the same as ours, and her method tolerates the same types of failures. In her scheme, the information concerning object location is distributed across the system. Each node is required to have information about the objects that are resident at that node and may have information about objects at other nodes. The information about objects at other nodes is not guaranteed to be up-to-date. It only reflects the location of an object from some time in the past. If a user cannot find an object, it can ask a node to conduct a search for the object. The node receiving the request starts by asking its "neighbors," a set of nodes with which it can communicate. Two search methods are given. The first method is a centralized search in which the node first asked to conduct the search does all of the querying of other nodes until it finds the information it wants. The second method is a decentralized search in which each queried neighbor conducts a search of its neighborhood before replying. The centralized search is easier to control, but the decentralized search may be faster since neighborhood searches are conducted in parallel.

The original intent of this work was to determine if the time for the search could be traded off against storage space requirements of standard replication of the service state. The conclusion of Henderson's thesis was that searching for objects still required nodes to remember fairly large amounts of information to prevent anomalies caused by race conditions. The information is needed to keep searches from doing redundant work, missing an object that has moved to a node that has already been searched, or resurrecting deleted objects. Although the amount of information kept may not be as much as in a replicated state scheme, in a network of arbitrary topology the cost of the search outweighs any savings in storage space that there might be.

Note that in Chapter 5 we concluded searching might be a feasible way of finding the location service itself after it has moved. It is still the case that searching is expensive (although we are keeping track of only one "object," the configuration of the service, so there is not that much information), but we expect that most clients will not need to actually do a search to find the service so the amortized cost is low. In other words, we would be using searching as a last effort when more reasonable strategies failed.

6.2.3 Establishing a rendezvous

Mullender and Vitanyi [22] propose to model object finding as a match-making service that matches an object with its users[2]. In this model, an object posts its location at some number of nodes and a user queries some number of nodes for that the object's location. When a user finds a node that knows the object's location, a *rendezvous* has been established. Strategies for posting and querying range from objects posting at only one node and users doing a complete search if not all objects post to the same node (or users querying the centralized server if all objects do post to the same node) to objects posting at all nodes and the users just waiting for the information to arrive. Various strategies for posting and searching are described in their paper [22]. In the "shotgun" scheme, an object posts its location at a random set of nodes, and a user queries a random set of nodes. The idea is to choose the sets such that there is a high enough probability that a rendezvous will be made. In the "hash" scheme, both sets consist of the same nodes and are determined by a hashing function on the object's name.

It is not clear from their scheme what happens when an object wants to post at a node that is inaccessible to that object. For the shotgun scheme, it may not matter, and the object could just choose another node to post at until it posted at a pre-determined minimum number of nodes, but for the hash scheme, it is important that the information be posted at the particular node. If they expect to do a write-all update, then this scheme is clearly not as available as our method. The authors also do not address what happens if a user gets an out-of-date address. This could happen in the shotgun scheme since the nodes are chosen at random for both posting and querying. In the hash scheme, this could happen if they do not use a write-all update. In addition, they

[2]In their system, the objects that move are processes called servers, and the entities that look for them are called clients of these processes.

do not discuss whether a user is guaranteed to make progress if it queries again after getting an out-of-date address. There is a trade-off between the efficiency and availability of lookups versus the efficiency and availability of updates in this scheme. Posting at more nodes would mean that updates are less efficient, but that lookups are more available and in the case of the shotgun scheme, more efficient since it is more likely that a rendezvous will be established earlier. Posting at fewer nodes has the opposite effect.

The hash scheme is considered to be more efficient, but not as available as the shotgun scheme. The hash scheme is more efficient because the nodes to post and query are known ahead of time, so there is greater likelihood of establishing a rendezvous. However, this also means that if all of the specific nodes go down at the same time, there will be no way to locate the object in question. The authors also point out that the hash function would have to be reprogrammed in order to take any new nodes into consideration as rendezvous nodes if we want to keep the load approximately balanced.

It should be noted that our method can be described by this model. In our method, all objects (eventually) "post" at a fixed set of nodes. This set is also known by all users and can be queried by them. In effect, this makes these nodes "rendezvous" servers. We pick a particular posting strategy that is described by the multipart timestamp technique. We also specify what happens when clients receive out-of-date addresses and guarantee progress toward finding the current address.

6.3 Future work

An immediate area of future research that suggests itself is being able to provide a binding mechanism for handlers of different types. For example, we might like to replace a handler with one that takes more or fewer arguments or one that has arguments of a different type. The interesting part of this problem would be how to keep the information that tells how to match the original arguments to the arguments of the actual call. That is, instead of just keeping the path to the current location, each edge can also have a transformation function that is applied to the arguments of the call to the previous address to give the arguments for the call to the next address on the chain.

Another area of interest is to implement a server using the alternate strategy of delaying the answering of queries if the query needs to have certain types of information instead of delaying the update operations. This would probably not be a feasible idea for the location service since lookups are expected to be much more frequent than rebinds, but for services where the updates are expected to be more frequent, this might be a better way of solving ordering problems.

A third area of research is to find ways of reducing the size of the *send_gossip* list. For example, we could gossip only the update records of operations invoked at the particular replica. This would require changes in the gossip processing algorithm because the gossip messages no longer satisfy the prefix property that all operations included in the state associated with a sender's timestamp are included in its gossip or are known to have propagated to the receiver. We can no longer just merge the gossip message's timestamp into the replica's timestamp since the replica may not know all of the operations known by the sender. More importantly, if we do not send all known operations in gossip messages, our ability to process updates like the *rebind* operation that rely on the effects of other operations already being present is affected. The update record would have to record this dependency, and the gossip processor might have to wait for the needed operation(s) to arrive.

Another area of interest is to extend the reconfiguration scheme to allow reconfigurations that overlap. By overlap, we do not mean that reconfiguration transactions run concurrently, but that a reconfiguration transaction will be allowed to proceed even if the previous reconfiguration has not finished (that is, all of the information from the first configuration has not necessarily propagated to the second configuration). The difficult part of this extension is what to do if the change configuration information from the second transaction arrives at a non-participant replica before the change configuration information from first transaction.

A final area of future work is optimizing the gossip sending algorithm. The described algorithm has a worst-case performance of sending $O(n^2)$ messages, where n is the number of replicas. For small n, this may be tolerable, but if we have a very large system, n may be fairly large as well. For large n, we might use a hierarchical scheme where replicas gossip to a small group of replicas and only a few replicas from each group gossip to other groups. Another way to reduce the number of messages sent is to have replicas send acknowledgments of receipt.

References

[1] Bloom, T.
Dynamic Module Replacement in a Distributed Programming System.
Technical Report MIT/LCS/TR-303, MIT Laboratory for Computer Science, Cambridge, MA, March, 1983.

[2] Eswaran, K., *et al.*
The Notion of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-633, November, 1976.

[3] Fischer, M. and Michael, A.
Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network.
In *Proceedings of the Symposium on Principles of Database Systems, Los Angeles, California.* ACM, March, 1982.

[4] Fowler, R.
Decentralized Object Finding Using Forward Addresses.
Technical Report 85-12-1, University of Washington, Department of Computer Science, Seattle, WA, December, 1985.

[5] Gifford, D.
Weighted Voting for Replicated Data.
In *Proceedings of the 7th Symposium on Operating System Principles, Pacific Grove, California.* ACM, December, 1979.

[6] Gray, J.
Notes on Database Operating Systems.
Lecture Notes in Computer Science. Vol. 60 : Operating Systems, An Advanced Course.
Springer-Verlag, New York, 1978, pages 393-481.

[7] Henderson, C.
Locating Migratory Objects in an Internet.
Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, August, 1982.
Available as Computation Structures Group Memo 224, MIT Laboratory for Computer Science.

[8] Herlihy, M.
Replication Methods for Abstract Data Types.
Technical Report MIT/LCS/TR-319, MIT Laboratory for Computer Science, Cambridge, MA, May, 1984.

[9] Herlihy, M. and Liskov, B.
A Value Transmission Method for Abstract Data Types.
ACM Transactions on Programming Languages and Systems 4(4):527-551, October, 1982.

[10] Ladin, R., Liskov, B., and Shrira, L.
A Technique for Constructing Highly-Available Distributed Services.
MIT Laboratory for Computer Science, Cambridge, MA.
June, 1987
Submitted for publication.

[11] Lamport, L.
Time, Clocks, and the Ordering of Events in a Distributed System.
Communications of the ACM 21(7):558-565, July, 1978.

[12] Lampson, Butler W. and Sturgis, Howard E.
Atomic Transactions.
Lecture Notes in Computer Science. Vol. 105 : Distributed Systems-- Architecture and Implementation.
Springer-Verlag, New York, 1981, pages 246-265.
This is a revised version of Lampson and Sturgis's unpublished paper, "Crash Recovery in a Distributed Data Storage System".

[13] Liskov, B.
Overview of the Argus Language and System.
Programming Methodology Group Memo 40, MIT Laboratory for Computer Science, Cambridge, MA.
February, 1984

[14] Liskov, B., *et al.*
Abstraction Mechanisms in CLU.
Communications of the ACM 20(8), August, 1977.

[15] Liskov, B., *et al.*
Lecture Notes in Computer Science. Vol. 114: CLU Reference Manual.
Springer-Verlag, New York, 1981.

[16] Liskov, B., *et al.*
Argus Reference Manual.
Programming Methodology Group Memo 54, MIT Laboratory for Computer Science, Cambridge, MA.
March, 1987

[17] Liskov, B. and Guttag, J.
Abstraction and Specification in Program Development.
MIT Press, Cambridge, MA, 1986.

[18] Liskov, B. and Ladin, R.
Highly-Available Distributed Services and Fault Tolerant Distributed Garbage Collection.
In *Proceedings of the 5th Symposium on Principles of Distributed Computing, Calgary, Alberta, Canada.* ACM, August, 1986.

[19] Liskov, B. and Scheifler, R.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
ACM Transactions on Programming Languages and Systems 5(3):381-404, July, 1983.

[20] Marzullo, K.
Loosely-coupled Distributed Services: a Distributed Time Service.
PhD thesis, Stanford University, Stanford, CA, 1983.

[21] Moss, J.
Nested Transactions: An Approach to Reliable Distributed Computing.
Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, Cambridge, MA, April, 1981.

[22] Mullender, S. and Vitanyi, P.
Distributed Match-Making for Processes in Computer Networks - Preliminary Version.
In *Proceedings of the 4th Symposium on Principles of Distributed Computing, Minaki, Ontario, Canada.* ACM, August, 1985.

[23] Oki, B.
Reliable Object Storage to Support Atomic Actions.
Technical Report MIT/LCS/TR-308, MIT Laboratory for Computer Science, Cambridge, MA, May, 1983.

[24] Parker, Jr., D., *et al.*
Detection of Mutual Inconsistency in Distributed Systems.
IEEE Transactions on Software Engineering SE-9(3):240-247, May, 1983.

[25] Schlichting, R. and Schneider, F.
Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems.
ACM Transactions on Computer Systems 1(3):222-238, August, 1983.

[26] Walker, E.
Orphan Detection in the Argus System.
Technical Report MIT/LCS/TR-326, MIT Laboratory for Computer Science, Cambridge, MA, June, 1984.

[27] Weihl, W.
Distributed Version Management for Read-only Actions.
IEEE Transactions on Software Engineering, Special Issue on Distributed Systems SE-13(1):55-64, January, 1987.

[28] Weihl, W. and Liskov, B.
Implementation of Resilient, Atomic Data Types.
ACM Transactions on Programming Languages and Systems 7(2):244-269, April, 1985.

[29] Wuu, G. and Bernstein, A.
Efficient Solutions to the Replicated Log and Dictionary Problems.
In *Proceedings of the 3rd Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada.* ACM, August, 1984.

OFFICIAL DISTRIBUTION LIST

Director Information Processing Techniques Office Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	2 copies
Office of Naval Research 800 North Quincy Street Arlington, VA 22217 Attn: Dr. R. Grafton, Code 433	2 copies
Director, Code 2627 Naval Research Laboratory Washington, DC 20375	6 copies
Defense Technical Information Center Cameron Station Alexandria, VA 22314	12 copies
National Science Foundation Office of Computing Activities 1800 G. Street, N.W. Washington, DC 20550 Attn: Program Director	2 copies
Dr. E.B. Royce, Code 38 Head, Research Department Naval Weapons Center China Lake, CA 93555	1 copy